LARGE SCALE HIERARCHICAL K-MEANS BASED IMAGE RETRIEVAL WITH MAPREDUCE

THESIS

William E. Murphy, Second Lieutenant, USAF

AFIT-ENG-14-M-56

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-14-M-56

LARGE SCALE HIERARCHICAL K-MEANS BASED IMAGE RETRIEVAL WITH MAPREDUCE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

William E. Murphy, B.S.

Second Lieutenant, USAF

March 2014

AFIT-ENG-14-M-56

LARGE SCALE HIERARCHICAL K-MEANS BASED IMAGE RETRIEVAL WITH MAPREDUCE

William E. Murphy, B.S.
Second Lieutenant, USAF

Approved:

/signed/	4 March 2014
Maj Kennard R. Laviers, PhD (Chairman)	Date
/signed/	3 March 2014
Maj Brian G Woolley, PhD (Member)	Date
/signed/	6 March 2014
Douglas D. Hodson, PhD (Member)	Date

AFIT-ENG-14-M-56

Abstract

Image retrieval remains one of the most heavily researched areas in Computer Vision. Image retrieval methods have been used in autonomous vehicle localization research, object recognition applications, and commercially in projects such as Google Glass. Current methods for image retrieval become problematic when implemented on image datasets that can easily reach billions of images.

In order to process these growing datasets, we distribute the necessary computation for image retrieval among a cluster of machines using Apache Hadoop. While there are many techniques for image retrieval, we focus on systems that use Hierarchical K-Means Trees. Successful image retrieval systems based on Hierarchical K-Means Trees have been built using the tree as a Visual Vocabulary to build an Inverted File Index and implementing a Bag of Words retrieval approach, or by building the tree as a Full Representation of every image in the database and implementing a K-Nearest Neighbor voting scheme for retrieval. Both approaches involve different levels of approximation, and each has strengths and weaknesses that must be weighed in accordance with the needs of the application. Both approaches are implemented with MapReduce, for the first time, and compared in terms of image retrieval precision, index creation run-time, and image retrieval throughput. Experiments that include up to 2 million images running on 20 virtual machines are shown.

Table of Contents

Page

List of Figures

Figure Page

List of Tables

List of Abbreviations

LARGE SCALE HIERARCHICAL K-MEANS BASED
IMAGE RETRIEVAL WITH MAPREDUCE

I. Introduction

In Afghanistan alone, US Military Intelligence is collecting over 53 TB of data per day [11]. Much of this data is video from UAVs, satellite imagery, and image data from other sources. According to General Robert Kehler, commander of United States Strategic Command (USSTRATCOM) [11], "There is a gap between our growing ability to collect data and our limited ability to process that data, we are collecting 1,500 percent more data than we did five years ago. At the same time, our head count has barely risen." By developing a scalable MapReduce image retrieval system, we take a step towards closing the gap between the ability to collect data and the ability to process it.

Image retrieval can be described as selecting an image from a database of images, that is most similar to a particular query. Current smart phone applications use image retrieval techniques to enable a person to snap a picture of a scene or an object, and search a database in order to get information on that scene or object. The scenes and objects that can be recognized are limited by the scenes and objects that have been indexed in the database collection of images. As the number of objects and scenes recognizable by these applications are increased, so is the number of images that need to be placed in the database collection of images. Image retrieval techniques have also been used to recognize objects and scenes in video, by treating the video frames as images.

It is clear that database collections of images will continue to grow, and the techniques used to index and search these collections need to adapt to the growing size. The primary challenges posed by these expanding database image collections that necessitate a change

in current image retrieval techniques are: recognition performance, storage, scalability, and computational cost.

MapReduce [16] is a computing framework created by Google® that allows for large scale parallelization and data distribution across a cluster of machines. MapReduce directly addresses the problems of storage and computational cost by distributing storage and computation across many machines. With less pressure to make approximations to improve storage and computational cost, techniques with better recognition performance can be used. MapReduce has been used for parallelizing large scale image retrieval from databases with up to 100 million images [4, 32]. This thesis investigation extends that research by integrating additional retrieval techniques.

1.1 Overview

Most successful image retrieval techniques can be decomposed into three principal steps: feature extraction, database index creation, and image comparisons. These three steps are necessarily sequential. First, features are extracted from all database images. These extracted features are then used to create an index of the database images. This index is then used to efficiently compare a particular query image to the database images. Each of these steps can be performed in various ways, and the manner in which these steps are performed dictates the image retrieval method being used. Each of these steps are discussed briefly in the following sections. Finally, the techniques used to implement them in a MapReduce framework is explained.

1.1.1 Feature Extraction.

Feature extraction is an integral part in efficiently comparing images. When choosing how to extract image features, feature detection and description are two tasks that must be performed. As the names suggest, a feature detection algorithm detects features in an image, and a feature descriptor algorithm characterizes the detected feature, typically as

a vector of doubles. Techniques for performing each of these tasks are discussed in more detail in later sections.

An image feature, in this context, can be defined as a section of an image which displays different image properties than the areas immediately surrounding it. Depending on the type of feature detector being used, this section can be a single pixel, an edge, or any type of region. The image properties typically considered when searching for features include, but are not limited to, intensity, color, and texture.

The most important property of an image feature is repeatability [44]. When presented two different images of the same object, a repeatable feature extractor will yield a large number of the same features detected on the object from both images. It is important that features are detected consistently, even when an image has been blurred, re-oriented, rescaled, or transformed in any other manner. Repeatability is obtained in feature extraction algorithms using two different techniques: invariance and robustness. These techniques, and how they provide for reliability, is further discussed in Chapter II.

(a) Sample Image from Caltech Buildings Dataset

(b) MSER Features Approximated with Ellipses

Figure 1.1: Side-by-Side display of an image from the Caltech Buildings Dataset, before and after MSER feature detection.

For this research we use the Maximally Stable Extremal Region (MSER) [28] feature detection algorithm. MSERs are shown to be optimal [31, 19] for object detection. A MSER can be described as a connected area in an image that has an intensity that is either higher or lower than all of the pixels immediately surrounding it. The necessary difference in the intensity inside and outside the MSER, or the threshold, is a parameter set in the detection algorithm. This parameter is set such that there is an average of 300 features detected per image.

The Scale Invariant Feature Transform (SIFT) [27] feature description algorithm was used to describe the detected MSER features. An ellipse approximates the MSER region as shown in Figures 1.1b and 1.1a, and the SIFT descriptor characterizes the region into a 128-dimensional vector. SIFT provides for reliability by being invariant to image scale and rotation. SIFT also achieves reliability through robustness, by being desensitized across a large range of distortion, 3D viewpoint, noise, and illumination. SIFT descriptors were developed for the purpose of image matching. They were designed to be highly distinctive allowing for a single feature to be matched with another feature from a large database. SIFT features have been shown to be the most effective descriptors [30] for image matching and are used in most image retrieval and image matching problems relevant to this work [6, 36, 4, 2, 12, 32].

1.1.2 Database Index Creation.

Indexing the database image collection consists of organizing the features extracted from the database images in a manner to facilitate fast queries. There are two main approaches to database index creation: Full Representation (FR) and Bag of Words (BoW) [12]. Each of these approaches are outlined in the following sections.

1.1.2.1 Full Representation.

When using FR, all features extracted from the database images are explicitly stored in a data structure that can be efficiently searched. The data structure storing the database

image features serves as the database index. For this work, we use Hierarchical K-Means Clustering (HKM) [34] to build a tree which consists of database image features composing the leaf nodes, and cluster centers composing the internal nodes. In order to build the HKM tree, the K-Means clustering algorithm is performed at each level of the tree until a prescribed maximum depth is reached. In a study of the most widely used FR image retrieval techniques, HKM was shown to yield the fastest retrieval times as well as recognition performance within ten percent of the top performer for all datasets tested [5]. This reason along with HKM's apparent suitability to MapReduce programming is the motivation to use HKM for building the FR index.

1.1.2.2 Bag of Words.

When using the BoW approach, all features extracted from the database image collections are quantized into pre-determined bins, and occurrence count is stored in lieu of the database features being stored explicitly. In order to build an index using a BoW method, we first construct a visual vocabulary from a set of training data. In this research we use HKM to build this a hierarchical vocabulary tree as introduced by Nister and Stewenius [36]. All extracted database features are then classified into one of the leaf nodes in the vocabulary tree, and feature occurrences per image at every leaf node are stored in inverted files [35]. HKM vocabulary trees represent the state of the art among BoW image retrieval methods in terms of speed, and are among the top performers in terms of recognition performance [3, 36].

1.1.3 Image Comparison.

Extracting features from the database images and building image database index are both off-line, or preprocessing, steps. The final step of image retrieval, and the actual on-line task, is to match a presented query image to a particular image in the database. Finding this particular database image involves comparing the query image to all images in the database, and subsequently returning the best match. The method used to make these

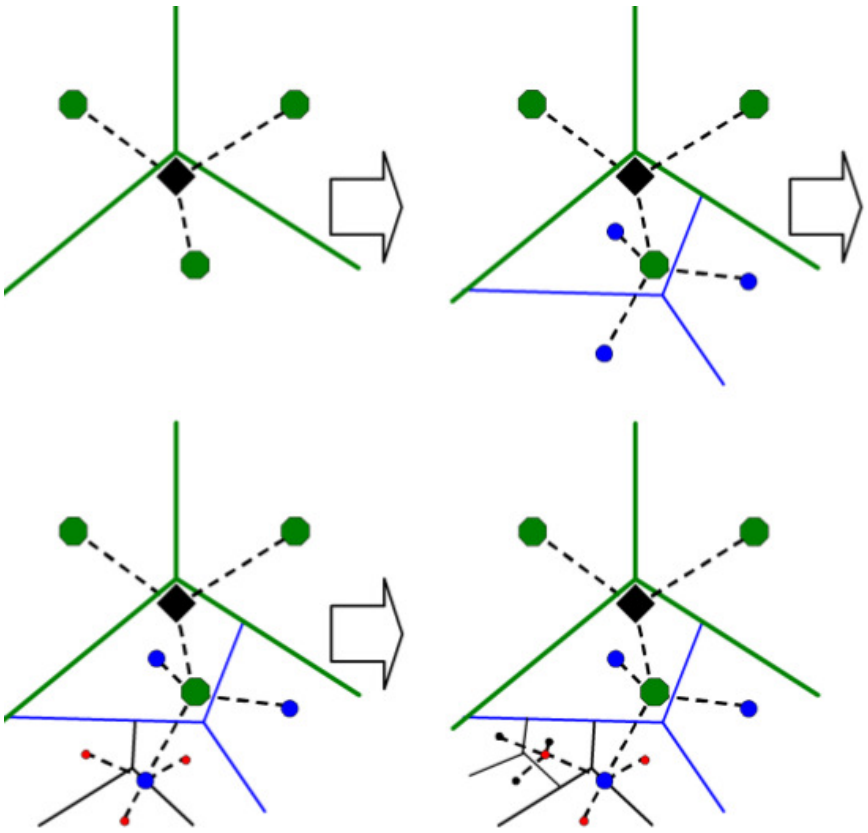


Figure 1.2: Visualization of HKM clustering process [36]. Here the branch is three, and K-Means has been performed on 4 levels. Note that the resultant children are only shown for one node at each level.

comparisons depends largely upon the type of database index that has been created. Aside from the index type, the first step is to detect and extract features from the query image. This is done using the same methods discussed regarding database images. The techniques used to compare a query image to the database images for both FR and BoW indexes are described briefly in the following sections.

1.1.3.1 Image Comparison for FR Index.

We use a K-Nearest Neighbor voting scheme [34], in order to retrieve the corresponding database image when presented a query image. In short, we find the k

database features closest in feature space to each query feature. Each database feature "votes" for the database image that it came from, and whichever image gets the most votes is declared the best match. This voting scheme is further explained in Chapter II. With an FR index, a nearest neighbor voting scheme is the standard method [2, 4, 5, 32].

1.1.3.2 Image Comparison for BoW Index.

Comparison for databases indexed using a BoW approach is more complex. In this case we use the scheme presented by Nister and Stewenius [36]. This scheme was shown to be superior when compared to other comparison approaches by Aly [3]. Each feature extracted from the query image is classified into one of the leaf nodes in the vocabulary tree. After this point an equation is used to assign an objective score to each database image explained in Chapter II.

1.1.4 MapReduce.

MapReduce [16] was created by Google for the purpose of parallelizing computation and distributing data across a large cluster of machines. When dealing with traditional parallelization, most of the code is written to deal with issues such as parallelizing computation, distributing data, handling potential node failures, and load balancing. MapReduce abstracts these issues and allows the programmer to focus on the actual task at hand. This is done by constraining the programmer to "map" and "reduce" functions.

MapReduce is a good solution for parallelizing data intensive problems. These types of problems involve processing and generating large datasets. Many data intensive problems are termed as "embarrassingly parallel", meaning that there is very little data dependency in the input data. Many parts of the image retrieval process are "embarrassingly parallel."

MapReduce was embraced for large scale applications by companies such as Yahoo!®, Facebook®, IBM® and Google®. At Google® alone, more than 100,000 MapReduce jobs

process more than 20 PB of data daily [16]. MapReduce is also a common way to leverage the potential of cloud computing [22].

The data processes and programming constraints set by MapReduce are important to consider because they dictate the way algorithms can be written, and the way data structures can be used. This becomes apparent when methodology and implementation are discussed in Chapter III.

1.2 Research Goals and Objectives

The goal of this research is to build two linearly scalable image retrieval systems. The techniques used to accomplish this goal are:

1. Create and implement MapReduce Hierarchical K-Means BoW and FR indexing algorithms
2. Develop MapReduce Query Image Comparison algorithms for BoW and FR Indexes
3. Analyze the scalability of each step in the image retrieval system.

Full testing and analysis is performed for these objectives to validate that the goal has been achieved. In order to test the linear scalability of the system, we examine the linear scalability of three component tasks in the system:

1. Index Storage Size
2. Index Creation Time
3. Query Image Throughput

We call the system linearly scalable if each of the tasks is linearly scalable. This methodology is explained in Chapter III.

1.3 Software System Design

When developing a software system, a top down approach is often used. In a top down design approach the developer establishes high level system requirements, and breaks the problem down into a series of ambiguous subsystems that contribute to fulfill overall system requirements. These subsystems are then refined to the point where the data structures and necessary operations are determined so that algorithms can be designed. Finally, the designed algorithms are implemented. The entire system design is shown in Figure 1.3.

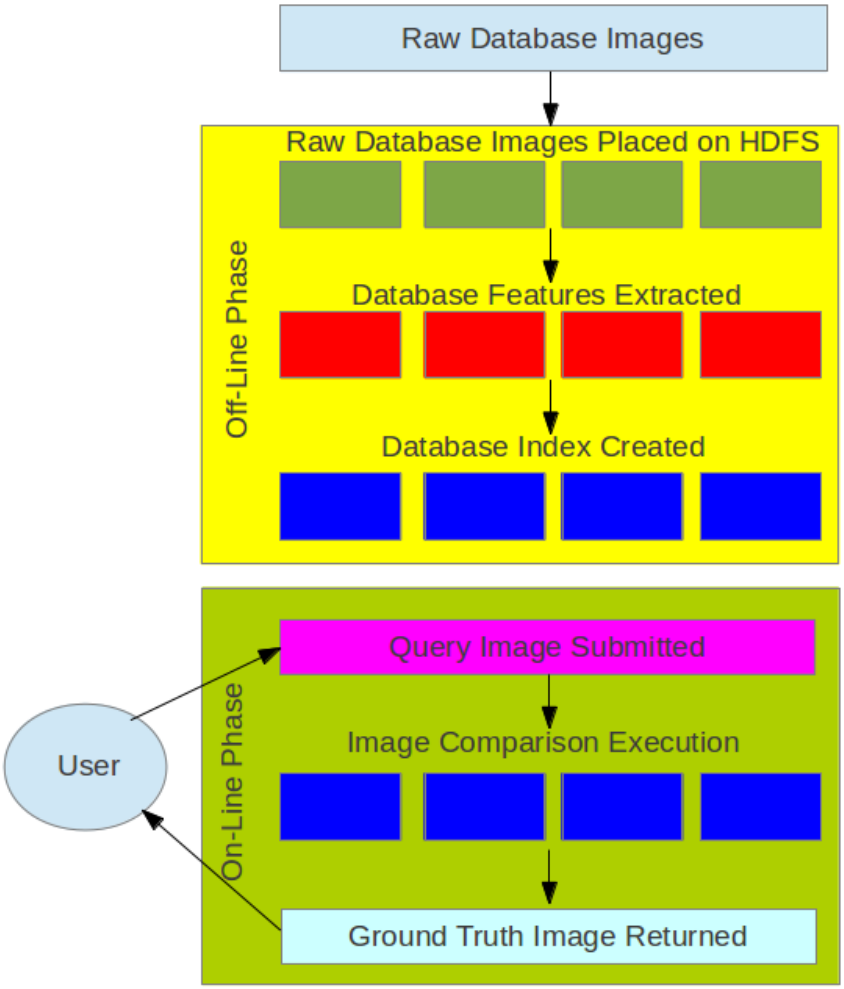


Figure 1.3: MapReduce Image Retrieval System Design

1.4 Assumptions

Image retrieval is a subcategory in the computer science field of computer vision. It is important that the reader uses this knowledge as context e.g. a feature in computer vision literature is not the same as a feature in artificial intelligence literature. Also, in order to reduce the scope of this work, it is assumed that the reader has a general knowledge of:

1. Calculus
2. Linear Algebra
3. Trigonometry and Geometry
4. Algorithms
5. Programming

Calculus, trigonometry, and geometry are fundamental prerequisites to understand feature detection and description. Basic linear algebra is used throughout this document to compare image features. Finally, a basic understanding of algorithms and programming is needed to understand the implementations described in Chapter III.

1.5 Risks

Implementing a virtualized MapReduce cluster on a single server is the most significant risk to this research. Limited resources and delayed recognition of this particular requirement left this as the only option. Another risk is the number of third party software packages being used. There are numerous potential problems associated with integrating the functionality of multiple software packages.

1.6 Thesis Outline

In Chapter II, background information related to image retrieval and large scale computation with MapReduce is discussed. Subjects forming the building blocks of this

research are discussed in detail, most importantly Aly's large scale Fully Represented image retrieval techniques [4], Nister and Stewenius's [36] vocabulary tree approach, and Dean and Ghemawat's introduction of MapReduce [16]. Past research along with new techniques presented in this document serve as the foundation for the design and implementation detailed in Chapter III. The implementation is tested and results are presented for tests over multiple database sizes are analyzed in Chapter IV. Finally, the work is concluded, and possible extensions to this research are suggested in Chapter V.

II. Background and Related Work

To understand the discussion of a large scale image retrieval system built with MapReduce, the reader must have knowledge of: fundamental image retrieval and MapReduce. These two concepts are used in this research to produce an image retrieval system powerful enough to accept very large database image collections. A full explanation of MapReduce and it's open source implementation, Hadoop [47], is given in Section 2.1. A background on image retrieval techniques and their underlying concepts is presented in Section 2.2. Finally Section 2.3 presents prior MapReduce image retrieval systems. While some of these topics are briefly discussed in the previous chapter, more comprehensive and technical explanations are given here.

2.1 MapReduce and Hadoop

When attempting to process enormous amounts of data in parallel, even the most simple tasks require large amounts of complex code. This is due to the inherent difficulties of writing parallelized code. When dealing with traditional parallelization, most of the code is written to deal with issues such as parallelizing computation, distributing data, potential node failures, and load balancing. MapReduce abstracts these issues and allows the programmer to focus on the actual task at hand. This is done by constraining the programmer to ”map” and ”reduce” functions. There is also an optional ”combine” function provided for optimization.

MapReduce is software that runs on each machine in a cluster of machines, accompanied by a programming model. By following the MapReduce programming model, the programmer specifies to the software how the job should proceed, and the software directs the job execution until completion.

Programs written using the MapReduce programming model are automatically parallelized on the cluster of machines they are set to run on. The MapReduce runtime handles input data partitioning, program execution scheduling, machine failures, and inter-machine communication. MapReduce clusters can be scaled to thousands of machines allowing very large datasets, without requiring the programmer to change their code. The details of the MapReduce programming model and software are explained in the sections below.

2.1.1 Hadoop and HDFS.

Hadoop [47] is an open source implementation of MapReduce along with the Hadoop Distributed File System (HDFS) [9]. This is the implementation of MapReduce used in this research. In this document the terms Hadoop and MapReduce are used interchangeably.

2.1.1.1 HDFS.

HDFS is a distributed file system, based on the Google File System (GFS) [20], that provides centralized access to all files stored on the Hadoop cluster. HDFS is implemented at the user level with Java. This means that when deploying HDFS on a cluster, no changes need to be made to the native kernel implemented file system currently running on each machine in the cluster. HDFS runs on top of the native kernel based file systems on each node, and utilizes them to store data. It is not necessary to run the same type of local file system on each node [39], making HDFS extremely portable. Data stored on HDFS is broken into blocks, 64 MB by default, and each block is stored as a separate file on HDFS. These files are then each replicated to two separate nodes in order to protect from node failures.

Two types of Java services compose HDFS, namely the DataNode and the NameNode. The DataNode service runs on all nodes in the Hadoop cluster. The DataNode serves as the interface between the NameNode and the kernel implemented file system on each machine.

The DataNode performs basic operations on the data being stored on the local machine which it is running with the direction of the NameNode.

The NameNode service runs only on the ”master” node of the Hadoop cluster. This centralized service is responsible for storing a directory of the data on the Hadoop cluster. This directory contains a list of all of the files on HDFS, and all of their locations and it is stored in memory. In order to perform basic data operations such as open, delete, or rename on HDFS, a client is directed to the NameNode. The NameNode responds to the client with the locations of the data, and the client interacts with the DataNode on the nodes specified by the NameNode.

2.1.1.2 HDFS Data Representation.

Data representation is an important consideration when processing large amounts of data using Hadoop. The problem is especially important when dealing with a large number of relatively small files [18]. The first issue with having a large number of small files on HDFS, is that the NameNode has memory limitations. Recall that the NameNode stores a directory of all files, and their locations in memory. The second issue with having a large number of small files, is that the total time to process them in batch is dominated by disk seeks. These issues are solved by combining these small files into larger types of files that follow the input/output data formats discussed below.

There are two principal input and output data formats that are supported by MapReduce. The simpler of the two formats is a text representation. In text mode, each line of the input is treated as a key-value pair. By default the key is the byte offset of each line, and the value is the actual contents of the line [47]. This is not very useful for many applications since it provides no flexibility in the potential key values. The key-value pair, in the text mode can also be separated by a user defined field delimiting character. This solves the problem of inflexible keys, however when the raw data is encoded into a value, the field and line delimiting characters must be avoided. The other commonly

used input type are SequenceFiles. This is the file format for all temporary outputs of map functions used within Hadoop. A SequenceFile is a flat file composed of key-value pairs, stored in binary. Hadoop provides tools for writing, reading, sorting, and compressing SequenceFiles. The SequenceFile format allows many images to be stored in their original binary form, to be stored in a single file. This is the way image data is represented when placed on HDFS in this research.

2.1.1.3 Hadoop Engine.

We refer to the software responsible for managing a job's execution the MapReduce Engine [39]. Like HDFS, there are two Java services that compose the MapReduce Engine: the JobTracker and the TaskTracker.

The JobTracker is a centralized service that runs on the "master" node of the cluster. The JobTracker service is the supervisor for the execution of a MapReduce job. This service is responsible for splitting the input data into blocks, scheduling independent map and reduce tasks on nodes where the data to be processed resides, scheduling backup tasks, and re-scheduling tasks in the event of node failure. Executing a map task on a node that the data to be processed in that map task resides provides data locality, reduces network traffic, and saves time. Scheduling backup tasks prevents "straggler" tasks from dominating the total execution time of the job. Backup tasks are only scheduled towards the end of a MapReduce job, and they have been shown to significantly reduce the time to complete large operations[16].

The TaskTracker service runs on every computer in the cluster, and is responsible for running the map and reduce tasks. The JobTracker assigns tasks to the TaskTrackers when the job is submitted. When a task is assigned, the TaskTracker creates a new instance of a Java Virtual Machine (JVM) to execute it, once a task is complete the TaskTracker deletes the JVM instance. Through the program execution, the TaskTrackers contacts the JobTracker to report task completions, and request new tasks.

2.1.2 MapReduce Programming Model.

Input in MapReduce programs are expressed as key-value pairs, and yield output is also expressed as key-value pairs. The majority of the processing is typically done in the Map function. The Map function takes the input key-value pairs, performs the processing written by the programmer, and yields an intermediate key-value pair. MapReduce then groups all intermediate key-value pairs with the same key, and sends them to the same reducer. The input to the Reduce function is an intermediate key, and an iterator of values associated with that key. The Reduce function typically merges the values, in a manner specified by the programmer, to produce the program's output.

As mentioned earlier, there is an option for the programmer to specify a Combine function to decrease the amount of data passed between mapper and reducer. If a Combine function is specified, the intermediate key-value pairs (mapper output) are merged, as specified in the Combine function, prior to being dispersed to the reducer. This can significantly reduce the amount of network traffic. The Combine function is not well documented within Hadoop, and it is not guaranteed that the Hadoop program execution will apply the combiner. With these problems in mind, Lin [25] implemented an in-mapper combiner design pattern. The in-mapper combiner defers emitting the intermediate key-value pairs until all input key-value pairs have been processed by the Map function. The intermediate pairs are stored in memory until a user defined close method is called. This close method allows the user to combine the data stored in memory, and emit the combined key-value pairs just before the JVM instance associated with this Mapper task is deleted. This allows greater user control in the combine phase, however it can only be used when all of the intermediate key-value pairs fit in memory.

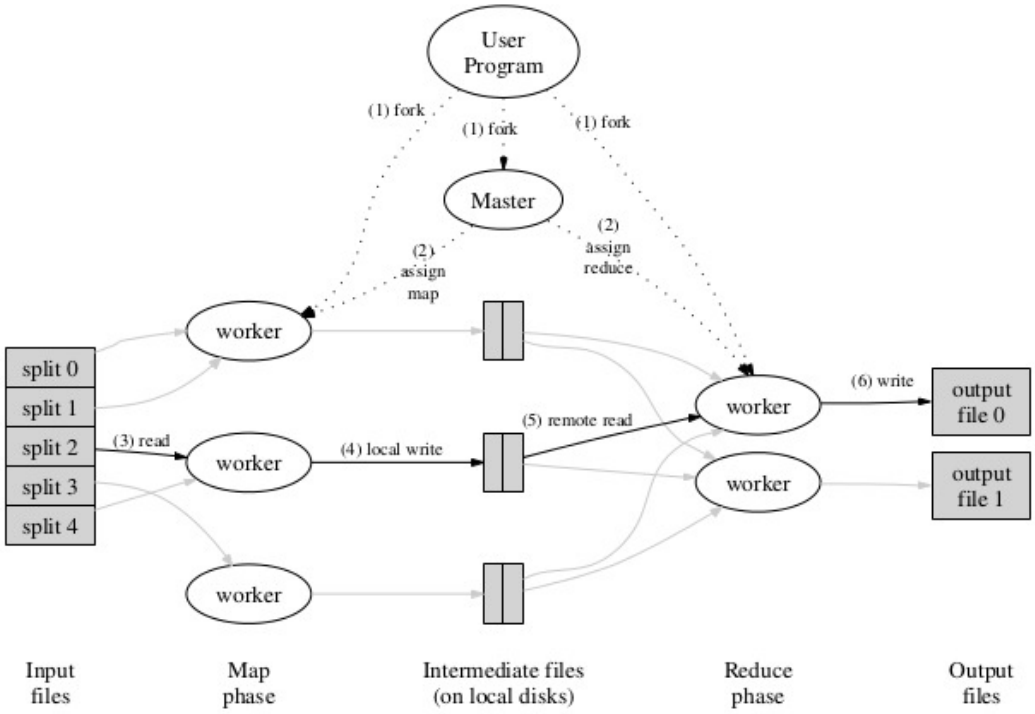


Figure 2.1: Data Flow for MapReduce Application [16]

2.1.3 Program Execution.

The steps for program execution in a MapReduce job can be seen in Figure 2.1 [16]. These steps are itemized as they are explained by Dean and Ghemawat in their unveiling of MapReduce [16] for clarity:

It is important to note that HDFS and the Hadoop Engine discussed above are already running at the start of this program execution. Also assume that the input data is already distributed on HDFS in 64MB blocks.

1. A User Program is submitted to the master node of the Hadoop cluster.

2. The JobTracker service on the master node, assigns map and reduce tasks to the TaskTracker services on idle worker nodes.

3. The worker nodes that have been assigned to run a map task read their corresponding input block, and parse the key-value pairs. The key-value pairs are then passed to the Map function, defined in the User Program.

4. The intermediate key-value pairs, or the outputs of the map function, are buffered in memory and periodically written to the local disk. The locations of these intermediate key-value pairs are passed to the JobTracker service on the master node, which is responsible for notifying the worker nodes assigned to run a reduce task.

5. When a reduce worker is notified of the location of the intermediate data by the master node, it reads the data from the map worker's disk. The reduce worker then sorts all of the intermediate data by key, so that all key-value pairs with the same key are grouped together.

6. The reduce worker then passes each intermediate key, and it's corresponding set of intermediate values to a separate instance of the user defined Reduce function.

7. The output of the Reduce functions from the same reduce worker are appended, and written to HDFS. The number of output files is the same as the number of reduce workers.

2.2 Image Retrieval

Before we continue the discussion of implementing a scalable image retrieval system with MapReduce, a technical background of image retrieval is needed.

2.2.1 Feature Extraction Background.

Feature extraction is an integral part in efficiently comparing images. An image feature, in this context, can be defined as a section of an image which displays different image properties than the areas immediately surrounding it. Depending on the type of feature detector being used, this section can be a single pixel, an edge, or any type of

region. The image properties typically considered when searching for features include, but are not limited to, intensity, color, and texture.

Among many other properties, the most important property of an image feature is repeatability. When presented two different images of the same object, a repeatable feature extractor will yield a large number of the same features detected on the object from both images. It is important that features are detected consistently, even when an image has been blurred, re-oriented, rescaled, or transformed in any other manner. Repeatability is obtained in feature extraction algorithms using two different techniques: invariance and robustness.

Invariance is achieved by mathematically modeling any expected transformations, and developing the feature extractor such that the expected transformations have no affect on the resulting features. Invariance is a good technique when significant transformations are expected. When the transformations are not expected to be large, robust feature extractors can be used to achieve reliability. Feature extractors that achieve repeatability through robustness are designed to be desensitized to the particular small transformations they are expected to encounter.

Some other important properties of good features according to a survey of features by Tuytelaars are:

- Distinctiveness: Intensity patterns used for detection should have a high variance.
- Quantity: A sufficient number of features should be detected even on small objects. The quantity of features detected should be easily throttled by a threshold value.
- Locality: The feature should be detected on a relatively small patch of the image.
- Accuracy: The features should be accurately localized on the image.
- Efficiency: Feature detection should be done as quickly as possible.

While these secondary properties of image features are important some are in place largely to support the primary effort of repeatability. Compromises need to be made when considering which properties are most important for a particular application. For example, distinctiveness and locality are competing properties. This is easily demonstrated when we consider the most localized feature possible, a single pixel. When considering a single pixel, it cannot be distinguished from another pixel of the same intensity, making it not very distinctive. On the other hand, if a feature is not localized it will be very distinctive due to a large area covered and a large amount of information in the feature, however it will be very susceptible to occlusions and other transformations. Invariance and Robustness also are competing properties to distinctiveness. As features are desensitized to large transformations (invariance) and information is thrown away (robustness), the features get less distinctive.

It is clear from the above discussion that the application must be considered when choosing the properties necessary to select a feature type. For example, in a laboratory setting where large transformations and other disturbances can be avoided, a very distinctive feature would be chosen, since reliability will not be severely affected by a low level of invariance and robustness. These properties are driven by both the feature detection and description algorithms.

2.2.1.1 Feature Detector Overview.

Feature detectors can be broken down into general categories: corner detectors, blob detectors, and region detectors. Well known detectors from each category are itemized below:

Corner Detectors

- Harris Detector [21]
- SUSAN Detector [43]

- Harris-Affine Detector [29]

Blob Detectors

- Hessian Detector
- Hessian-Affine Detector [29]
- Salient Regions [24]

Region Detectors

- Intensity-based Regions [45]
- MSER Detector [28]
- SuperPixels [33]

The detectors listed above are only a small fraction of the detectors that have been developed over time. These particular detectors are singled out because they have been used in image retrieval or object recognition research. While it is outside the scope of this research to explain the mathematical methods behind each of these detectors, the detector used in this work is explained along with the reason it was selected.

For this research we use the Maximally Stable Extremal Region (MSER) [28] feature detection algorithm. MSERs were shown to be optimal [31, 19, 13] for object detection, and were used in [36, 42]. The informal explanation of the process for finding MSER descriptors as given by Matas [28] is given here. First, we iterate through all possible thresholds of a gray-scale image. At each threshold, set the pixels with a gray level below the threshold to black, and the pixels with a gray level above the threshold to white. Now display the pixels with their white or black values, and step through the thresholds, we begin with a solid white image. As the threshold increases, black spots and regions begin to appear. All connected components, white and black, at each threshold are the maximal

regions. The MSER features are the regions that remain connected components over a large range of thresholds. MSER allows features to be excluded based on maximum and minimum area, and range of thresholds. The properties these regions exhibit that make them good features are listed below:

- Invariant to affine transformation of image intensities
- Stability: gives robustness to changes in lighting and noise from camera variation
- Multi-Scale detection: This gives locality in features that is important to avoid occlusions, as well as distinctiveness that is important for feature matching
- Regions can be detected in O(nlog(logn)), where n is number of image pixels

2.2.1.2 Feature Descriptor Overview.

There are also many potential feature descriptors to choose from. A list of the most common feature descriptors is given below:

- SIFT [27]
- SURF [8]
- HOG [14]
- Spin Images[23]

The Scale Invariant Feature Transform (SIFT) [27] feature description algorithm was used to describe the detected MSER features in this research. SIFT was shown to be the optimal descriptor for the MSER [13]. Implementations for describing an MSER region with a SIFT descriptor is given by Nister [36], and Dahl [13].

An ellipse approximates the MSER region as shown in Figure 1.1b, and the SIFT descriptor characterizes the region into a 128-dimensional vector. SIFT provides reliability

by being invariant to image scale and rotation. SIFT also achieves reliability through robustness, by being desensitized across a large range of distortion, 3D viewpoint, noise, and illumination. SIFT descriptors were developed for the purpose of image matching. They were designed to be highly distinctive allowing for a single feature to be matched with another feature from a large database. They have been shown to be the most effective descriptor [30] for image matching. SIFT descriptors are used in most image retrieval and image matching problems relevant to this work [6, 36, 4, 2, 12, 32].

2.2.2 Hierarchical K-Means Algorithm.

Prior to discussing the BoW and FR image retrieval techniques, both of which involve Hierarchical K-Means, it is important to explain the Hierarchical K-Means Algorithm. HKM is an extension on Lloyd's Algorithm [26], commonly known as k-means clustering. K-means clustering is used to categorize n observations into k clusters, by placing each of the n observations into the cluster with the nearest mean. The mean of each cluster, also referred to as the cluster center, is initialized by randomly selecting k observations from the set of n observations. Once all n observations are classified into one of the k clusters, the cluster centers are recalculated. This process is iterated until the observations all remain in the same cluster. In HKM, each of the k clusters output by Lloyd's Algorithm on the initial set of observations are again subjected to Lloyd's Algorithm.

Figure 2.2 is used for clarity. Initially, all n observations reside in the root node, or Node 0 as labeled in Figure 2.2. Lloyd's Algorithm is applied at Node 0, and each of the n observations are classified into either Node 1,2, or 3. Next, Lloyd's Algorithm is applied at Node 1, and each of the observations residing on Node 1 are classified into either Node 4,5, or 6. Loyd's Algorithm is also applied at Nodes 2 and 3, and all n observations are classified into leaf level nodes. At each level the clusters are split, and more granularity is achieved.

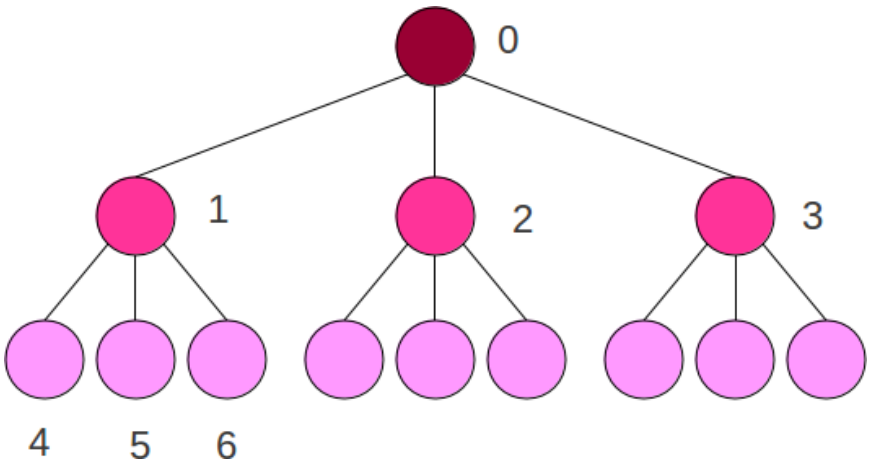


Figure 2.2: Illustration of a generic tree with branch factor=3, depth=2, and nodes numbered.

2.2.3 Bag of Words Retrieval Background.

The Bag of Words method for object and scene recognition was introduced by Sivic and Zisserman [42], with their "Video Google" software. When an image of an object or a scene from a movie in the "Video Google" database is submitted as a query, "Video Google" returns the frames that contain that object or scene. This is the same problem as the image retrieval problem discussed in this research. Their approach was influenced by text retrieval methods [7], and remains largely unchanged. The steps for their scene and object retrieval system are shown below:

- Detect MSER features from database frames, and describe them with SIFT descriptors.
- Build a visual vocabulary, using standard flat k-means, from a subset of the detected database features.
- Create a visual index, using vector quantization to count frequencies and create an Inverted File index.

- Use a Term-Frequency Inverse Document Frequency (tf-idf) weighting scheme, along with a normalized scalar product of two vector represented histograms in order to rank database frames

"Video Google" began a movement in image retrieval. Most research moved to a Bag of Words type method [12, 38, 48, 15]. In 2011, Aly conducted a benchmarking test [3] on the leading BoW methods. This study aimed to identify both the most efficient type of vocabulary, and the most efficient type of index to use. He benchmarks the Min-Hash index along with the Inverted File index. An inverted file stores, for each visual word, the images that contain them and how many times they occur in that image. The Min-Hash index creates hash functions for each image based on the features present in it. He tests both of these index types against a HKM vocabulary tree [36], and an Approximate K-Means (AKM) [38] index implemented with kd-trees. Both of these were developed as improvements on "Video Google". They attempted to improve retrieval performance by increasing the vocabulary size, and exploiting fast nearest neighbor searches. Aly determined that Inverted Files are better indexes for object retrieval in terms of retrieval times and accuracy. He determined that HKM trees provide retrieval times 10 times faster than AKM trees, but their accuracy is slightly worse.

The MapReduce BoW image retrieval implementation is largely based on Nister and Stewenius's methods using a HKM vocabulary and an Inverted File index, therefore a formal explanation is shown here starting at the point of constructing the visual vocabulary.

Nister and Stewenius use a vocabulary tree that is built by HKM clustering. A subset of the detected database features are used for creating the vocabulary tree. As opposed to the flat vocabulary used by Sivic and Zisserman [42], where k defines the number of quantization cells, here k defines the branch factor, of the tree. Initially, k-means is run on the entire training set, defining k cluster centers as children of the root node. Each feature in the training data is then classified into one of the k clusters represented by the

newly defined cluster centers, based on which cluster center it is closest to. This process is then continued for each new cluster of features down the tree until a maximum depth of L is reached. This process is shown in Figure 1.2. Once this process is complete, the vocabulary tree is defined, and the Inverted Files can be built. In order to build the Inverted Files, each database feature is propagated down the tree, being compared to k cluster centers at each level. This means that there are kL comparisons made. Each leaf node has an inverted file associated with it that stores the image id of each image with at least one feature through that node, and a count of how many features per image go through that node. The inverted file information of internal, non-leaf, nodes is simply a concatenation of the inverted files of the nodes beneath them.

Once the inverted files are built, Nister and Stewenius address how to quantify the relevance of a databse image to a query image. They define the scoring by assigning a weight w_i, to each node i in the vocabulary tree based on entropy. Entries in the query q_i vectors and database d_i vectors are then defined as:

$$q_i = n_i w_i \tag{2.1}$$

$$d_i = m_i w_i \tag{2.2}$$

where n_i and m_i are the number of descriptor vectors of the query and database image, respectively, with a path through node i. The entropy based weighting w_i at each node is given by:

$$w_i = \ln \frac{N}{N_i} \tag{2.3}$$

Where N is the total number of database images, and N_i is the number of database images with a path through node i. This is known as tf-idf scoring, and is a concept carried

over from text retrieval techniques. This weighting scheme provides less consideration to nodes that are represented by a large percentage of the database images.

A relevance score s between a particular database image and a query image is then found by calculating the normalized difference between the query and database vectors defined in Equations 2.1 and 2.2. This is given by:

$$s(q,d) = \| \frac{q}{\|q\|} - \frac{d}{\|d\|} \| \tag{2.4}$$

When Equation 2.4 is evaluated in L_2 norm,it simplifies to:

$$\|\mathbf{q} - \mathbf{d}\|_p^p = 2 - 2 \sum_{i|q_i \neq 0, d_i \neq 0} q_i d_i \tag{2.5}$$

Nister and Stewenius emphasize that the largest contributor to image retrieval performance is vocabulary size, and the leaf nodes are much more powerful than inner nodes. Nister and Stewenius implement stop lists, assigning a weight of 0 to nodes represented by many images. Use of stop-lists was shown to decrease matching time, with no negative attempts to performance.

The MapReduce implementation of BoW image retrieval is based on this research, however significant algorithmic redesign is needed in order to fit the MapReduce programming model. The MapReduce implementation is shown in Chapter III.

2.2.4 Full Representation Retrieval Background.

Full Representation (FR) is a simpler, less approximated method of image retrieval. The term Full Representation was, to the best of our knowledge, coined by Aly [5]. The approach however, was introduced by Lowe [27] as a practical use for SIFT. The general steps for FR image retrieval, excluding feature extraction steps, are itemized below:

1. Process and store database features in some data structure according to the image retrieval method being used.

2. Search the database feature data structure for nearest neighbors to each query image feature.

3. Assign a score to database images based on nearest neighbor search results according to method being used.

4. Find best matches by sorting database images based on score.

As can be seen from the steps above, Full Representation image retrieval is essentially an application approximate nearest neighbor search. Fast Library for Approximate Nearest Neighbors (FLANN [34] is a popular library for approximate nearest neighbor search introduced by Muja and Lowe. In a large study of many popular techniques for approximate nearest neighbor search, Muja and Lowe found that one of two algorithms achieved the best performance, depending on the dataset. These two algorithms are HKM and Randomized Kd-Trees. These are the same two algorithms that were found optimal for building the vocabulary in BoW image retrieval. Both algorithms are explained below, in the context of approximate nearest neighbor search with emphasis on the HKM algorithm. In the following paragraphs, whenever distance is mentioned, it refers to the Euclidean distance between two vectors given by:

$$\|\mathbf{q} - \mathbf{p}\| = \sqrt{(\mathbf{q} - \mathbf{p}) \cdot (\mathbf{q} - \mathbf{p})} \tag{2.6}$$

Randomized kd-trees were introduced by Silpa-Anan and Hartley [41] in 2008. To build a randomized kd-tree, the data is split in half at each level of the tree. The dimension on which to split the data is randomly chosen from the first D dimensions on which the data has the greatest variability. In order to search the tree, a priority queue is created with lowest bin distance first, and largest bin distance last, where the bin distance is the distance from a particular query feature to the bin center characterizing a leaf node of the kd-tree.

A fixed number of leaf nodes are searched, and a fixed number of nearest neighbors are returned.

The HKM algorithm for approximate nearest neighbor search was improved by Muja and Lowe [34]. The creation of the HKM tree is very similar to the creation of the vocabulary tree as described in the previous section. However, instead of quantizing the database features into a leaf node, and creating an Inverted File, the leaf nodes here are composed of a collection of database features. Muja and Lowe concluded that when performing k-means clustering at each level of the tree, 7 iterations are enough to get 90% of the nearest neighbor performance. This reduces the creation time to only 10% of the time required when run to convergence at each iteration. In order to search the HKM tree, each query feature is propagated down the tree into the cluster centered closest to that particular query feature. A priority queue is kept, and the unexplored paths down the tree are added to the priority queue in accordance with the query feature's distance to the cluster at the end of the path. A fixed number of leaf nodes are searched for nearest neighbors, and a fixed number of nearest neighbors are returned.

After the nearest neighbor search, the only remaining task to complete image retrieval is the k-nearest neighbor voting scheme. The k nearest neighbors, returned from the database, to each query image feature "vote" for the database image that they came from. The database images are then sorted, and the image with the largest number of votes is considered the best match.

2.3 Previous Large-Scale and MapReduce Image Retrieval Systems

Aly implemented a [4] distributed FR Kd-Tree algorithm for image retrieval with MapReduce. Experiments were run with database sizes up to 100M images and using a 2048 node cluster. Recall that kd-trees and HKM trees are both considered state of the art for FR image retrieval. Although this work is not done on the same scale as the work done by Aly, it does compliment the large scale work done using kd-trees.

White [46] introduced implementations of basic image retrieval algorithms on MapReduce. These algorithms, and implementations were used as building blocks for this work. White implemented both a flat k-means clustering algorithm, and a BoW algorithm using a flat vocabulary on MapReduce.

In 2013, Moise and Shestakov [32, 40], have been researching large scale indexing and search with MapReduce. They implement a system using an extended Cluster Pruning algorithm for indexing [32], and an ambiguous hierarchical clustering algorithm [40]. This hierarchical clustering algorithm is explained as HKM with randomly initialized centers, with one iteration performed. This is interesting as the index creation time will be greatly reduced, however image retrieval performance will almost certainly suffer. Moise and Shestakov ran tests with 100M images on 108 nodes.

III. Design and Implementation

When developing a software system, a top down approach is often used. In a top down design approach the developer establishes high level system requirements, and breaks the problem down into a series of ambiguous subsystems that contribute to fulfill overall system requirements. Image retrieval systems are very well suited to the top down approach since the problem can be broken down into very modular sequential tasks.

While these are the modular tasks that must be accomplished, they are governed by the requirement to work well with large scale datasets. The requirement of processing large datasets drove the system design to a MapReduce implementation. These tasks are then refined to the point where the data structures and necessary operations are determined so that the algorithms can be designed. Finally the designed algorithms are implemented, tested, and used in an empirical evaluation.

3.1 Test Environment

3.1.1 Cluster Hardware.

MapReduce was created to take advantage of the aggregate computational power of large numbers of commodity machines. Unfortunately, no physical clusters of commodity machines were available for this research. VMware virtualization software is used to simulate a commodity cluster on a single server. The single server has 2 Intel Xeon 5520 processors, which are specialized for virtualization, 200GB of memory, and 12TB disk space. The server is running VMware ESXi 5.0.0. VMware vSphere is used to manage a virtual cluster of 20 virtual machines, each with 9GB of memory, and 200GB of disk space. The Ubuntu Server 12.10 operating system is installed on each virtual machine in the cluster. The maximum number of nodes is limited by the available RAM for on the server. The number of virtual machines that compose the cluster is varied with experiments. While

a virtualized environment on a single server would not be an ideal hardware implementation for a production cluster, it serves the purpose for this research.

3.1.2 Cluster Software.

Many third party software packages are used in the implementation of this software system. The software packages, and their role in this research are:

- Apache Hadoop 1.2.1 [47]: This was the most current stable version of Hadoop at the time of this research. Hadoop is the open-source implementation of MapReduce and HDFS.
- OpenCV [10]: C++ library for Computer Vision tasks. OpenCV's implementation of MSER features and SIFT descriptors were utilized.
- Hadoopy [46]: Python wrapper for Hadoop. Hadoopy allows for Hadoop programs to be written in Python, and gives easy access to useful Hadoop functions. Used for creation of TypedBytes Sequence Files.
- Boost.Python[1]: Allows for wrapping C++ code to create Python modules. Exposes full C++ OpenCV libraries, as well as native speeds for computation intensive functions.

3.1.3 Datasets.

The vast majority of the images used for this research are from the ImageNet [17] dataset. The ImageNet dataset is a collection of over 14 million images divided into over 20 thousand subsets. For this research, these images serve as confusion images. In other words, these images only provide filler for the image database, and no attempt is made to retrieve them.

Query datasets are used in order to quantify the retrieval precision. In this research, the Caltech Buildings dataset [6] and the benchmark dataset used in [36] are used as

query datasets. The Caltech Buildings dataset includes images of 50 different buildings around the Caltech campus. There are five images of each building taken from all different positions. The UK Benchmark dataset is a collection of 2550 different household and office objects. Each object was photographed with four different orientations.

Figure 3.1: Caltech Buildings Dataset Sample

From each image category in the query datasets, one of the images is selected as a ground truth image. This ground truth image is inserted into the database among all of the confusion images. The remaining images in each category of the query datasets serve as the query images. These images are used to query the database with the expectation that the ground truth image from their respective category is returned.

Figure 3.2: UK Benchmark [36] Dataset Sample

3.2 Implementation

The implementation discussed here, is the first contribution of this thesis. Algorithms and their relations to data structures, are rethought in order to efficiently fit the MapReduce programming model. The restructuring of traditional image retrieval techniques to fit the MapReduce programming model, is the primary method used to accomplish the objectives of this research. Pseudo code is shown in the following sections to demonstrate the details of algorithmic restructuring for MapReduce implementations.

As discussed earlier in this document, there are three principal tasks that need to be implemented for an image retrieval system. First, an image retrieval system must extract features from a large database of images. Next, the database features must be indexed to enable fast search. Finally the system must accept query images, and search the database

for ground truth images associated with the query images. Two design approaches are taken for both the indexing and the database search tasks, and a comparison between the two is done.

3.2.1 HDFS Data Representation.

Before Hadoop can be used to begin the image retrieval process, the image database must be placed on HDFS. Recall from Chapter II that the images must be repackaged into sequential binary representation of key-value pairs, or a TypedBytes SequenceFile. This is made easy with the HDFS function implementations in Hadoopy. The ”writetb” function in Hadoopy, takes two inputs: an HDFS filepath, and a Python Iterator of key-value pairs. The key-value pairs chosen to get the database imagery onto HDFS are:

- **key**: ImageName (string)
- **value**: ImageData (binary image data)

The Python Iterator is written to include all databse images. After executing this function, the entire database image collection is placed on HDFS in TypedBytes SequenceFiles, each with approximately 1000 images. This is not a MapReduce job, it is executed from the client machine upon which it was called.

3.2.2 Feature Extraction in MapReduce.

As is described in Chapter II, feature detection involves first detecting features in the set of database images, and then describing the detected features.

This is a Map only job, and is a prime example of an “embarrassingly parallel” task. There is no Reduce function because in feature detection, there is no data dependency between the images and therefore nothing to be gained by aggregating the output of the Map function in any way. The lack of data dependency between the images in this task indicates that the amount of parallelization possible for this is limited only by the size of our cluster. The “detectFeatures” and “extractDescriptors” functions were implemented in

function MAP(*key*, *value*)

▷ key: string clusterID

▷ value: (string ImageName , Mat ImageData)

$MSERfeatures \Leftarrow detectFeatures(ImageData)$

$SIFTdescriptors \Leftarrow extractDescriptors(MSERfeatures)$

for feature **In** SIFTdescriptors **do**

Yield $('0', (ImageName, feature))$

end for

end function

Figure 3.3: MapReduce Feature Extraction Algorithm

C++ using the OpenCV libraby. The C++ implementation is wrapped with Boost.Python, so that upon compilation a Python module with calls to the C++ functions was created. Note that the key value being yielded by this Map function. The string "0" represents cluster "0", or the root node of the tree.

3.2.3 Index Creation in MapReduce.

The next step in creating a MapReduce image retrieval system, is creating an index of the database features. MapReduce is well suited for indexing large datasets. In fact, Inverted File creation for text documents was used by Dean and Ghemawat [16], to showcase the usefulness of MapReduce. Google uses MapReduce to produce the data structures used to index web pages for performing Google searches [16]. We implement two indexing methods: an Inverted File index built from an HKM vocabulary tree for BoW image retrieval, and an HKM tree for FR image retrieval.

The MapReduce implementation of HKM is used for building both types of indexes. HKM is used in BoW retrieval for building the visual vocabulary tree, and in FR the HKM

tree serves as the index. The Map, Combine, and Reduce Functions for the MapReduce Implementation of Lloyd's Algorithm are shown in Figures 3.4, 3.5, 3.6, respectively. This implementation is adapted from the k-means implementation given by White in [46]. Figure 3.7 shows the process of transforming this flat MapReduce k-means algorithm into a hierarchical application.

function MAP(*key*, *value*)

▷ key: string clusterID

▷ value: (string ImageName , vector feature)

$clusterCenters \Leftarrow$ **Load**($clusterCenters.pkl$)

$newCluster \Leftarrow assignCluster(clusterCenters[clusterID], feature)$

Yield ($newCluster$, ($ImageName$, $feature$)

end function

Figure 3.4: MapReduce K-Means Algorithm Map Function

Prior to the first iteration of the k-means algorithm, recall that the k cluster centers must be initialized. These cluster centers are chosen randomly from the set of data to be clustered, and written locally to a python dictionary data structure. The cluster centers are written as the value, with the cluster ID as the key. This dictionary is serialized using the Pickle Python module, and loaded into memory on each Mapper node in the Hadoop cluster. The "assignCluster" function is implemented in C++ with the Boost.Python [1] wrapper to create a Python module. This function takes in a feature, as well as the k candidate cluster centers to which this feature can be assigned. The Euclidean distance between the feature and each candidate cluster center is calculated using Equation 2.4. The candidate cluster center with the smallest Euclidean distance from the feature is returned, and the feature is assigned to that cluster.

```
function COMBINE(key, values)
▷ key: string clusterID
▷ values: Iterator (string ImageName , vector feature)
    count ⇐ 0
    featureSum ⇐ 0
    for (ImageName,feature) In values do
        count+ = 1
        featureSum+ = feature
    end for
    Yield (clusterID, (featureSum, count))
end function
```

Figure 3.5: MapReduce K-Means Algorithm Combine Function

Before passing the intermediate data to Reducer nodes, all key-value pairs with the same keys are combined by the Combine function shown in Figure 3.5. The Combine function takes in the output of the Map function, and sums features that share a common cluster and keeps track of how many have been summed. These partial sums and counts are then yielded, and there is only one output for each cluster per Mapper. It is easy to see how the Combine reduces network traffic and execution time.

Finally, in the Reduce function shown in Figure 3.6, the new cluster centers are calculated. The Reduce function takes in the output from the Conbine function. All partial sums with the same cluster Id are summed, and their associated partial counts are summed. The final sum is divided by the final count to give the cluster center. The yielded key-value pairs from the Reduce function are written to HDFS, as the final output of a single iteration

```
function REDUCE(key, values)
▷ key: string clusterID
▷ values: Iterator (vector featureSum , int count)
    finalCount ⇐ 0
    finalFeatureSum ⇐ 0
    for (featureSum,count) In values do
        finalCount+ = count
        finalFeatureSum+ = featureSum
    end for
    newCenter ⇐ finalFeatureSum/finalCount
    Yield (clusterId, newCenter)
end function
```

Figure 3.6: MapReduce K-Means Algorithm Reduce Function

of k-means. These key-value pairs are read from HDFS, and the local dictionary of cluster centers is updated with this information.

Figure 3.7 shows the full process for HKM in MapReduce. The functions shown in Figures 3.4, 3.5, and 3.6, represent a single MapReduce program. They perform a single iteration of k-means on an entire level of a tree in parallel. For succinctness in Figure 3.7, this program will be referred to as "kMeans Full Iteration".

The tree is built level by level starting at the root node and moving down. It is important to realize that the "kMeansFullIteration" MapReduce program performs an iteration of k-means on the entire current level of the tree. This is made clear by stepping through the algorithm to the second level. At the root level, the only node is the root node and all of the features are in the root node ('0'), see Figure 3.3. Initial cluster

```
1:  function FullProcess
2:      for level < maxDepth do
3:          for node On currentLevel do
    ▷ Choose k random features from each node for initial cluster centers
4:              Write(clusterCenters.pkl)
5:          end for
6:          for iteration < maxIterations do
7:              MapReduce:kMeansFullIteration
8:              Update(clusterCenters.pkl)
9:          end for
10:         MapReduce:kMeansMapFunction
11:         Delete:previousIterationData
12:     end for
13: end function
```

Figure 3.7: Full Process for MapReduce Hierarchical K-Means Algorithm

centers are chosen as cluster centers for the k children nodes of the root node. Next, the "kMeansFullIteration" MapReduce program is run, and the cluster centers are refined. Finally, the "kMeansMapFunction", shown in Figure 3.4 is run to classify all features into the new clusters, based on the new refined cluster centers, and the old HDFS cluster data is deleted. Now, all data on HDFS is classified in clusters ['00','01',....,'0k']. Back to the beginning of the algorithm, we choose k random features from each of these clusters to represent the cluster centers of each of their k children. The "kMeansFullIteration" program then refines all of these cluster centers, and the algorithm continues until the tree's prescribed maximum depth is reached.

The process shown in Figure 3.7 shows the process for creating a HKM tree. The vocabulary tree used for creating the Inverted File for BoW image retrieval is simply the dictionary stored in the "clusterCenters.pkl" file. When using this algorithm to build the vocabulary tree, the depth of the tree is chosen by the user based on the number of "visual words", or leaf nodes, desired. The number of leaf nodes is given by:

$$numWords = k^L \tag{3.1}$$

where k is the tree branch factor, and L is the depth of the tree. When building the tree or FR retrieval, the maximum depth of the tree is reached when one of the clusters contains k or less features. This is checked during the Reduce function of the "kMeansFullIteration" program. When this condition is met, the FR HKM index is built by running the Reduce function shown in Figure 3.8, as the Reducer for the "kMeansMapFunction" that is run on line 10 of Figure 3.7.

```
function REDUCE(key, values)
▷ key: string clusterID
▷ values: Iterator (string ImageName, vector feature)
    features ⇐ []
    for (ImageName,feature) In values do
        features.append[(ImageName, feature)]
    end for
    Yield (clusterId, features)
end function
```

Figure 3.8: Reduce Function for Building FR HKM Index

Figure 3.8 shows the function that runs as the complimenting Reducer for the ”kMeansMapFunction” that is run on line 10 of Figure 3.7 as soon as one of the clusters contains k features or less. This amount of granularity was shown to be optimal [34] The result is a list containing tuples of (ImageName, feature) associated with each cluster, and is the FR HKM Index.

While the algorithm shown in Figure 3.7 creates the necessary vocabulary tree, another MapReduce function is needed to create the Inverted File for BoW image retrieval. This function is shown in Figures 3.9 and 3.10.

```
function MAP(key, value)
▷ key: string clusterID
▷ value: (string ImageName , vector feature)
    clusterCenters ⇐ Load(clusterCenters.pkl)
    for level < maxDepth do
        newCluster ⇐ assignCluster(clusterCenters[clusterID], feature)
        clusterId ⇐ newCluster
    end for
    Yield (clusterId, ImageName)
end function
```

Figure 3.9: MapReduce Map Function for Creating Inverted File

In Figure 3.9 the “clusterCenters.pkl” file serves as the vocabulary tree. Each feature from the entire set of database features is classified into the leaf node with the closest cluster center, using the “assigncluster” function explained above in a for loop. Once the feature is assigned a cluster, we no longer need the feature data because the Inverted Files only store

```
function REDUCE(key, values)
▷ key: string clusterID
▷ values: Iterator(string ImageName)
    dict ⇐ emptyDictionary
    for ImageName In values do
        dict[ImageName]+ = 1
    end for
    Yield (clusterId, dict)
end function
```

Figure 3.10: MapReduce Reduce Function for Creating Inverted File

occurrences per image, per leaf cluster. The Reduce function, shown is Figure 3.10 defines the Inverted File using a dictionary. Each time an image name occurs in the output of the Map function, it is counted as a tally. For each clusterId, a dictionary is built storing the feature occurrences per image.

3.2.4 Scoring and Retrieval in MapReduce.

The final step after creating an index, is using that index to score database image relevance against a query image, and return the most relevant images. While the previous steps are done off-line in a preprocessing phase, this step is an on-line phase. The first part of this phase involves detecting, and classifying features in the query images. This is done locally, since the query datasets are relatively small. The "clusterCenters.pkl" file created during the index creation stage for both types of indexes, is used to classify the query features. A dictionary is created and stored locally to represent the query data. This query data dictionary is then serialized to a Pickle file named "queryData.pkl" for use on the Hadoop cluster. This query data dictionary contains the feature vector, the query image

from which the feature came, and the cluster to which the feature was assigned. By storing the query image, we are able to compare many query images to the database images in parallel. The algorithm for locally creating the query data dictionary is shown in Figure 3.11.

```
function ProcessQueryImages
    clusterCenters ⇐ Load(clusterCenters.pkl)
    queryImages ⇐ Load(/QueryImageFile)
    queryData ⇐ emptyDictionary
    for image In queryImages do features ⇐ detectFeaturesimage
        for feature In features do
            clusterID ⇐ assignCluster(clusterCenters, feature)
            if clusterID In queryData.keys then
                queryData[clusterID].append((queryImageName, feature))
            else
                queryData[clusterID] ⇐ [(queryImageName, feature)]
            end if
        end for
    end for
end function
```

Figure 3.11: Processing Query Images for FR Image Retrieval

Now that the features in the query images have been extracted and classified, comparisons to database images can be made. The methods for BoW and FR are both explained below. It is important to note that the BoW retrieval technique uses the Inverted

File index, and the FR retrieval technique uses the FR HKM tree index. Figures 3.12 and 3.13 shows the MapReduce k-nearest neighbor voting program for FR image retrieval.

```
function MAP(key, value)
▷ key: string clusterID
▷ value: (string databaseImageName , vector databaseFeature)
    queryData ⇐ Load(queryData.pkl)
    if clusterID In queryData.keys then
        for (queryImageName,queryFeature) In queryData[clusterID] do
            dist ⇐ EuclideanDist(queryFeature, databaseFeature)
            Yield (queryImageName, (queryFeature, databaseImageName, dist))
        end for
    else
▷ DoNothing
    end if
end function
```

Figure 3.12: Map Function for FR Query/Database Image Comparison

Figure 3.12 is the Map function for the program that uses a k-nearest neighbor voting scheme for making comparisons between the database and query images in a FR MapReduce image retrieval system. The Map function takes in a single key-value pair from the FR HKM tree index. If there are any features from query images that reside in the same cluster as this database feature from the FR HKM index, the distance between the database feature and the query features are calculated. If there are no features from query images that reside in the same cluster as this database feature, we do nothing and move on to the next key-value pair from the FR HKM database.

```
function REDUCE(key, values)
▷ key: (string queryImageName
▷ values: Iterator(vector queryFeature, string databaseImageName, double dist)
    topMatches ⇐ emptyDictionary
    intvotesPerQueryFeature
    for ( do(queryFeature,databaseImageName, dist) In values
        if length(topMatches[queryFeature])<=votesPerQueryFeature then
            topMatches[queryFeature].append((databaseImageName, dist))
            sort(topMatches[queryFeature])                ▷ ascending sort on dist
        else
            if dist < topMatches[queryFeature][votesPerQueryFeature - 1] then
                delete topMatches[queryFeature][votesPerQueryFeature − 1]
                topMatches[queryFeature].append((databaseImageName, dist))
                sort(topMatches[queryFeature])            ▷ ascending sort on dist
            end if
        end if
    end for
    votingData ⇐ emptyDictionary
    for databaseImage In topMatches.values do
        votingData[databaseImage]+ = 1
    end for
    Yield (queryImageName, votingData)
end function
```

Figure 3.13: Reduce Function for FR Query/Database Image Comparison

Figure 3.13 shows the Reduce function for the program that uses a k-nearest neighbor voting scheme for making comparisons between the database and query images in a FR MapReduce image retrieval system. This function takes in the intermediate data from the Map function shown in Figure 3.12, and reorganizes the data in order to tally the votes based on the distances between query and database features. A dictionary is built with the count of votes for each database image with a minimum of one vote, when compared to the query image named in the input key. The database images with the highest number of votes are considered the best matches for the query image named in the input key. A key-value pair is written to HDFS with a voting data dictionary for each query image.

Scoring and Matching with the Inverted File, requires a different MapReduce program. Similar to FR, we need to detect and classify features in order to use the Inverted Files for BoW image retrieval. Instead of storing the features along with the cluster to which they have been classified, we only store occurrences per cluster, per image. This processing of the query images is done with the same program used for building the inverted files of the database images. The algorithms that compose this program are shown in Figures 3.9 and 3.10. The Inverted Files characterizing the query images are read from HDFS, and written to a local dictionary with the keys being cluster Ids, and the values being lists of tuples of the form ccurrences) where "occurrences" is the number of features from the query image of n(queryImageName, name "queryImageName" that were quantized into the cluster corresponding to the key clusterID. This local dictionary is serialized to a Pickle file called "queryIF.pkl".

To score the database images relevance to the query images, we refer back to Equation 2.5, where q_i and d_i are defined in 2.1 and 2.2 respectively. The Map function for comparing database and query images using the Inverted Files is shown in Figure 3.14. Here, we use the tf-idf weighting scheme, and we take advantage of the inverted file by only performing a multiplication, when both a query image and a database image have a feature in the

cluster at hand. While submitting multiple query images at the same time complicates the programs, extra complication reduces the amount of overhead and disk seek time incurred by the MapReduce program.

function MAP(*key*, *value*)

▷ key: string clusterID

▷ value: dict featureCount

score ⇐ *emptyDictionary*

$weight \Leftarrow ln\frac{N}{length(featureCount.keys)}$

queryIF ⇐ **Load**(*queryIF.pkl*)

if clusterID **In** queryIF.keys **then**

for (queryImageName, occurences) **In** queryIF[clusterID] **do**

for (databaseImageName) **In** featureCount.keys **do**

$partialScore \Leftarrow featureCount[databaseImageName] * weight * occurrences * weight$

Yield ((*queryImageName*, *databaseImageName*), *partialScore*)

end for

end for

else

DoNothing

end if

end function

Figure 3.14: Map Function for BoW Query/Database Image Comparison

function REDUCE(*key*, *values*)

▷ key: ((string queryImageName, string databaseImageName))

▷ values: Iterator(int partialScore)

$totalScore \Leftarrow 0$

for partialScore **In** values **do**

$totalScore+ = partialScore$

end for

$finalScore \Leftarrow 2 - 2 * totalScore$

Yield $(queryImageName, (databaseImageName, finalScore))$

end function

Figure 3.15: Reduce Function for BoW Query/Database Image Comparison

Figure 3.15 shows the Reduce function for comparing database and query images using the Inverted Files. Here we are combining the partial scores that are yielded from Map functions when a database and query image both have at least one feature in the cluster used as input to the Map function. Once the partial scores have been summed, the relevance score between the query and database images are computed by Equation 2.5. These scores are copied from HDFS, and a simple sort returns the order of the best matches.

3.3 Experimental Methodology

As is stated in Chapter I, the goal of this research was to design and implement a scalable HKM image retrieval system using MapReduce. Most of the problems that exist with current image retrieval systems are related to the scale of the database. Some of these problems are storage of database index, database index creation time, image retrieval throughput, and image retrieval accuracy. This research shows linear scalability in terms of

database feature storage, database index creation, and image retrieval throughput. We also discuss why this system is not scalable in terms of image retrieval accuracy.

There are two methods for demonstrating scalability. These methods are strong and weak scalability. Strong scalability involves showing how a parameter varies with number of computing nodes and fixed data size. While weak scalability involves showing how a parameter varies with number of computing nodes for a fixed data size per processor. For strong scalability to be considered linear, the chosen parameter must decrease at a rate equal to the rate of the increasing number of computing nodes. For weak scalability to be considered linear, the chosen parameter must remain constant as the data size increases at the same rate as the number of computing nodes. In this work, we focus on weak scalability. Scaling up the nodes in a cluster to meet the needs of increasing amounts of data is the most common use case for MapReduce, and therefore we use the scalability metric that directly characterizes that use case.

First, we examine the scalability of the database index storage. The scalability of the storage is unique because the storage size is not dependent on the number of computing nodes used to build it. As a result, the concepts of strong and weak scalability are not applicable. In order to show the scalability of the database index storage, we simply report the size of the FR HKM and Inverted File database indexes for various data sizes. Table 3.1 shows the tests that are run, in order to show scalability with respect to index storage. While the number of leaf nodes in the Inverted File affect the size of the index, no heuristic is known for choosing the number of leaf nodes. Therefore, the number of leaf nodes is held constant for each size data set at 1,000,000 leaf nodes. Another parameter that is held constant, that may also affect the outcome of the Inverted Index experiments is the percentage of database features used for training data. We use 10 percent of the database features for training data to build all Inverted Files.

Database Index Storage Scalability Tests	
Index Type	Number of Images
Inverted Files	500,000
FR HKM	
Inverted Files	1,000,000
FR HKM	
Inverted Files	2,000,000
FR HKM	

Table 3.1: Database Index Storage Scalability Tests: In each scenario, database index size is recorded.

Scalability is shown by examining the rate at which the database index size varies with the size of the data set. Linear scalability is achieved when the index size increases at a rate equal to the rate of the increase in data size. An increase in index size growth rate that is less than the rate of increase in data size, is better than linear and is highly desirable. A one-tailed z-test will be used in order to show with statistical significance whether the scalability of the index storage is linear or better.

Next, we test the scalability of the time to create the index. For the weak scalability test, we increase the data size with the number of cluster nodes. This allows us to examine how the time to create the database index varies with increasing cluster size, and fixed data size per cluster machine. Again, a one-tailed z-test will be used in order to show with statistical significance whether the scalability of the index creation time, for each type of index, is linear or better.

The final aspect of this system that we statistically exam, is the scalability of the retrieval throughput. The retrieval throughput is defined as the time taken per image to

Index Creation Time: Weak Scalability		
Index Type	Number of Cluster Nodes	Number of Images
Inverted Files	5	500,000
FR HKM		
Inverted Files	10	1,000,000
FR HKM		
Inverted Files	20	5,000,000
FR HKM		

Table 3.2: Index Creation Time: Weak Scalability

return the results of best matches in the database. We examine the weak scalability for the CalTech Buildings dataset presented earlier in this Chapter. The test procedure for examining the scalability of the retrieval throughput is shown in Figure 3.3. For the k-nearest neighbor search in the retrieval and comparison step of FR image retrieval, we set k equal to the branch factor of the tree. The only variance in the retrieval throughput, for both FR and BoW systems, is introduced through the variance in the index. In order to capture how the varying indexes affect the throughput, we test throughput on multiple FR and BoW indexes and present the results with a specified level of confidence.

The last piece of the system observed is the retrieval performance. Retrieval performance at n is defined as the percentage of correct ground truth images that are returned in the top n results. The retrieval performance is only presented with mean and standard deviation here. Retrieval performance is an artifact of the BoW and FR methods, and is not affected by this parallelization or implementation. It has been shown in all image retrieval systems, that as the size of the database image set increases the retrieval performance decreases. For each of the database sizes previously used and for each of

Scalability of Retrieval Throughput			
Dataset	Index Type	Number of Cluster Nodes	Number of Images
Caltech Buildings	Inverted Files	5	500,000
	FR HKM		
	Inverted Files	10	1,000,000
	FR HKM		
	Inverted Files	20	2,000,000
	FR HKM		

Table 3.3: Retrieval Throughput: Weak Scalability

the query datasets, we examine the percentage of times the correct ground truth image is returned in the top n results, over a range of values for n. The range of n values, used for each scenario is $n = [1, 10, 30, 50]$. In other words, we examine the percentage of cases in which the ground truth image is ranked, according to the score assigned in the retrieval and comparison step: as the top image, in the top ten images, in the top thirty images, etc.

IV. Results and Analysis

The results here are presented and analyzed in the order of the order of the testing procedures discussed in Chapter III. We begin by examining the database index storage scalability. The approximate sizes of the raw database image data sets, subsets of the imageNet [17] dataset, are shown in Table 4.1

Raw Database Image Size	
Number of Images	Size of Data
500,000	250GB
1,000,000	500GB
2,000,000	1TB

Table 4.1: Raw Database Image Size

We extract approximately 300 features from each database image. The total approximate number of features, the approximate size of the set of features, and the approximate time taken to extract features for each database image set is shown in Table 4.2.

The time to extract features is discussed in the index creation time section. The index storage size mean (μ) and standard deviation (σ) for the Inverted Files index, and FR HKM index are shown in Table 4.3. It is important to note the fixed parameters, training data and number of leaf nodes discussed in Chapter III. While these parameters would certainly have an affect on the index size, we hold them constant in order to isolate the affect that the growing data size has on the index size.

Total Approximate Feature Count and Feature Storage		
Number of Images	Number of Features	Size of Feature Set
500,000	150M	25GB
1,000,000	300M	50GB
2,000,000	600M	100GB

Table 4.2: Total Approximate Feature Count and Feature Storage

4.1 Index Storage Scalability

Index Storage Size			
Index Type	Number of Database Images	Mean Index Size (μ)	Standard Deviation (σ)
Inverted Files	500,000	2.62GB	0.00091
	1,000,000	5.13GB	0.0019
	2,000,000	9.41GB	0.0084
FR HKM	500,000	26.30GB	0.002
	1,000,000	51.72GB	0.0069
	2,000,000	103.02GB	0.013

Table 4.3: Total Index Storage Size

It is clear from the standard deviations associated with the index storage results that there is very little variance in this metric. This is simply because all of the information is being stored in each case. The small variance comes from the random initialization of the cluster centers, that can cause some features to end up in different leaf nodes from test to

test. This has a very small effect on the storage when an image previously unrepresented in a node becomes represented.

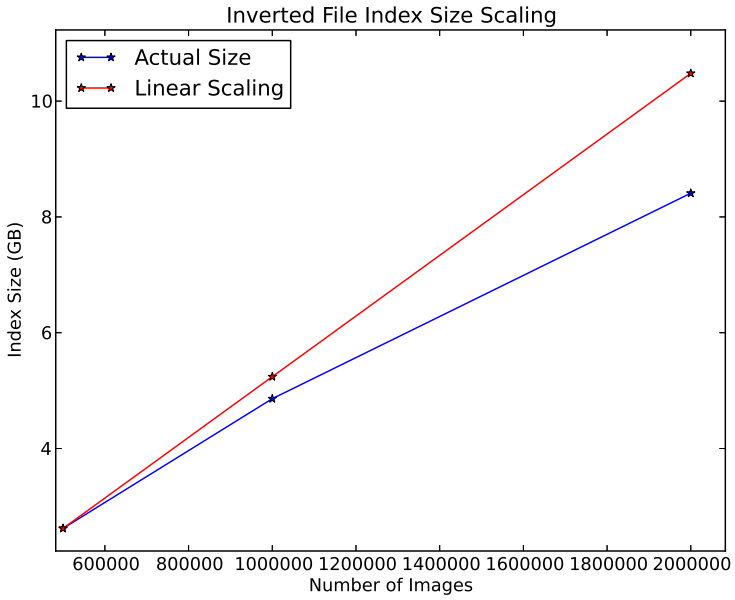


Figure 4.1: Inverted File Index Storage Scaling

Figure 4.1 shows graphically that the mean of the Inverted File Index Size scales better than linearly. We take the index size at 500,000 database images, and increase that value at the same rate as the number of database images increases to get the "Linear Scaling" line. It is clear that our Inverted File mean index size is growing at a lower rate than that of the database image collection. Here we use the one-tailed t-test to show statistical significance in the linear scalability. We perform the t-test in order to test the hypothesis that the index size is less than or equal to the maximum size allowed for linear scalability. The relevant equation for performing the t-test is shown in Equation 4.1.

$$t = \frac{\bar{X} - \mu_o}{\frac{s}{\sqrt{n}}} \tag{4.1}$$

where s is the sample standard deviation, n is the sample size, $\bar{X}$ is the sample mean, and μ_0 is the null hypothesis. Now we can set the null hypothesis to the maximum value size for linear scalability for 1,000,000 database images and 2,000,000 database images and determine the level at which we are confident that the index storage is at least linearly scalable. In order to calculate the confidence value, all of the inputs to the one-tailed t-test are plugged into Equation 4.1, and the t value is calculated. The t value, along with the Degree of Freedom (n-1), is used in a lookup table, and the confidence is given. Now that the t-test has been explained, further confidence levels are given along with the necessary inputs without explanation.

Inverted File Statistical Storage Scalability		
Number of Images	Inputs	Confidence
1,000,000	$\mu_0 = 5.24$	99.5%($t = -81.88$)
	$\bar{X} = 5.13$	
	$s = 0.0019$	
	$n = 2$	
2,000,000	$\mu_0 = 10.48$	99.75%($t = -180.14$)
	$\bar{X} = 9.41$	
	$s = 0.0084$	
	$n = 2$	

Table 4.4: Inverted File Statistical Storage Scalability

One interesting observation, is that the Inverted File index is roughly one order of magnitude less than the FR HKM index. Also, the FR HKM index is only slightly larger than the set of detected features in each database size. This is expected, since the FR HKM

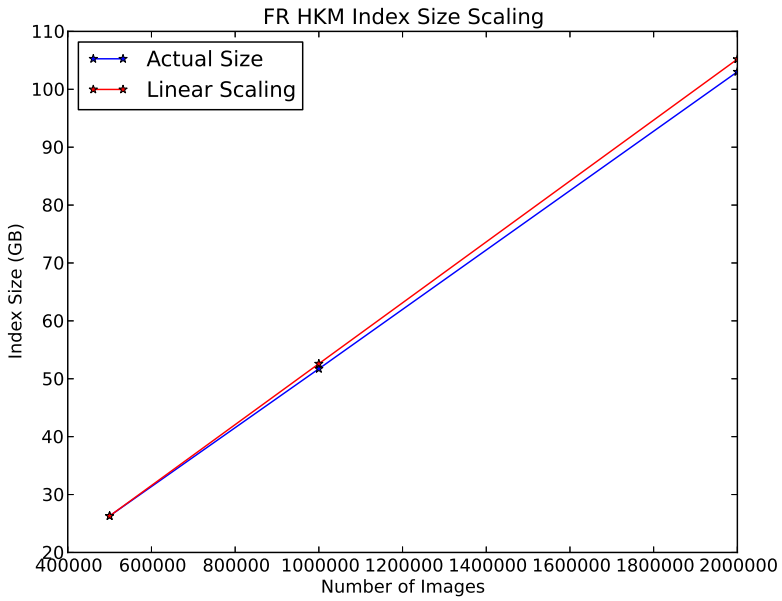


Figure 4.2: FR HMKM Index Storage Scaling

index is simply an organization of the detected features. The most interesting observation then, is that the Inverted Files index reduces the amount of storage necessary by a full order of magnitude when compared to the set of detected features. The figures and tables presented above, show that this image retrieval system is linearly scalable with respect to data storage. Next, we address the scalability with respect to index creation time.

4.2 Index Creation Time

Due to the ”embarrassingly parallel” nature of feature detection and extraction, it is assumed to be linearly scalable. Since there is absolutely no data dependency between the individual database images from which features are detected, there is nothing prohibiting linear scalability. In this section, we begin to time index creation, after detection and extraction of features.

Due to the iterative nature of HKM, it has very large index creation times. Index creation is a preprocessing step in image retrieval that is performed only one time, therefore

FR HKM Statistical Storage Scalability		
Number of Images	Inputs	Confidence
1,000,000	$\mu_0 = 52.6$	99.75%($t = -180.36$)
	$\bar{X} = 51.72$	
	$s = 0.0069$	
	$n = 2$	
2,000,000	$\mu_0 = 105.2$	99.75%($t = -237.15$)
	$\bar{X} = 103.02$	
	$s = 0.013$	
	$n = 2$	

Table 4.5: FR HKM Statistical Storage Scalability

large index creation times are generally accepted, if they lead to improvements in other areas. This testing shows differences in the index creation times of Inverted Files and FR HKM. First, we examine creation times for Inverted Files indexes. Again, all inverted files were built with 1,000,000 leaf nodes and 10% of the database image features are used for training.

Inverted File Creation Times			
Nodes in Cluster	Database Images	Mean Creation Time (μ)	Standard Deviation (σ)
5	500,000	174.1 min	2.37 min
10	1,000,000	176.36 min	2.26 min
20	2,000,000	175.08 min	2.04 min

Table 4.6: Inverted File Creation Times

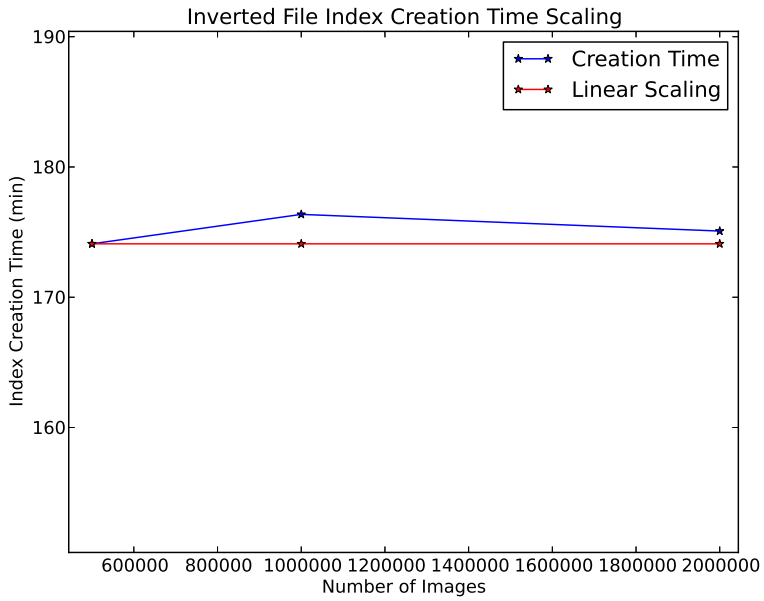


Figure 4.3: Inverted File Creation Time Scaling

It is clear from Figure 4.3 that we do not have linear scaling. This can be attributed to randomness in cluster initialization. If the random initialization chooses cluster centers that cause features to be distributed unevenly, we have a lack of load balancing. Underutilized nodes sit idle, while other nodes process more than their fair share and this is inefficient. While we cannot show that the Inverted File index creation time has linear or better weak scalability, we use the two-sided t-test to construct 95% confidence bounds to show how we can expect the weak scalability of the index creation. The resultant 95% confidence bounds are shown in Table 4.7.

These confidence bounds tell us that statistically we can be 95% confident that when we create a database index as described in this work, the time to create it will fall between the upper and lower limits. While not as strong a case as for index storage, these confidence bounds show that the Inverted File creation is approximately linearly weakly scalable.

IF Creation Times 95% Confidence Bounds			
Cluster Nodes	Database Images	Lower Limit	Upper Limit
10	1,000,000	172.76 min	179.96 min
20	2,000,000	171.83 min	178.33 min

Table 4.7: Inverted File Creation Times with 95% Confidence

Next, we examine creation times for FR HKM indexes. These indexes involve performing HKM on the entire dataset, as opposed to the training subset to create Inverted Files. This leads to longer index creation times, however it involves less approximation. The first attempt at implementing an FR HKM in this work was very inefficient. No Combiner was implemented, and therefore at the top level of the HKM tree, every one of the n features in the database was classified into k nodes. Not implementing a Combiner meant that approximately $\frac{n}{k}$ features were sent to only k Reduce tasks. After monitoring this implementation running for a week, we implemented the Combiner function, and the index creation time dramatically decreased.

FR HKM Index Creation Times			
Nodes in Cluster	Database Images	Mean Creation Time (μ)	Standard Deviation (σ)
5	500,000	1492.21 min	9.16 min
10	1,000,000	1513.4 min	10.26 min
20	2,000,000	1522.74 min	8.65 min

Table 4.8: FR HKM Index Creation Times

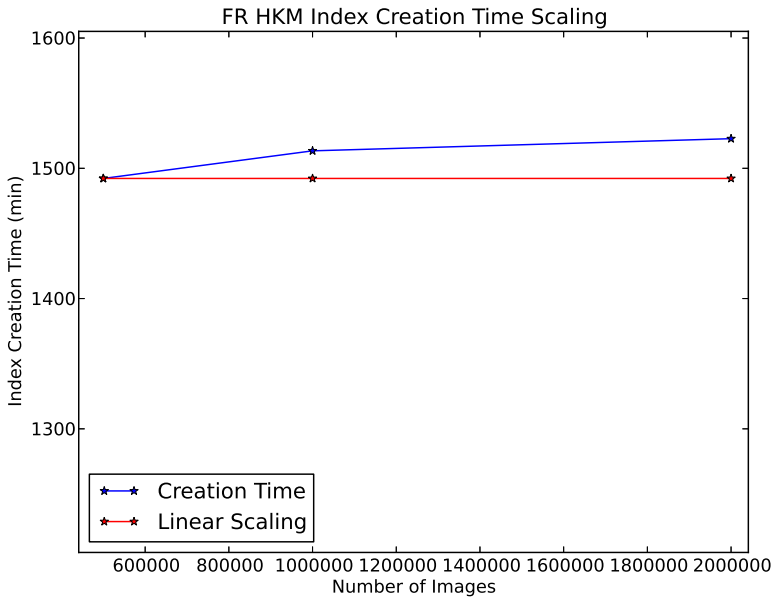


Figure 4.4: FR HKM Index Creation Time Scaling

As can be seen in Figure 4.4, again we cannot show that the creation times for FR HKM indexes are always linear or better. This happens due to the same randomness that caused the Inverted Files to diverge from linear weak scalability. In the FR HKM index creation, we get larger deviations because, we are clustering very large amounts of data. We again use the two tailed t-test in order to determine the region on which we are 95% confident.

FR HKM Creation Times 95% Confidence Bounds			
Cluster Nodes	Database Images	Lower Limit	Upper Limit
10	1,000,000	1487.91 min	1538.69 min
20	2,000,000	1501.25 min	1544.23 min

Table 4.9: FR HKM Creation Times with 95% Confidence

The 95% confidence bound (n=3) on the FR HKM index shows a non-linear weak scaling. This is most likely due to the built in scaling of the algorithm. The HKM tree automatically grows to more leaf nodes as the data grows, since the tree reaches maximum depth when a node contains less nodes than the branch factor of the tree. Even with the tree growing, the weak scalability is near linear which, in practice, is much more common than linear weak scalability. In the linear weakly scalable system, the amount of data per node stays constant as the amount of data and the number of nodes both increase. In this case as the number of nodes and the amount of data are increased, we see a decrease in the productivity of the cluster in terms of images indexed per second at each node.

4.3 Retrieval and Comparison Throughput

The retrieval and comparison throughput is defined as the number of images matched with the k best matches from the database per second. We observe the weak scalability of this throughput for both BoW and FR retrieval approaches using the CalTech Buildings query dataset. We start with BoW retrieval using the CalTech Buildings Dataset, which uses the Inverted Files index to search the database. Timing this process begins with query features being detected and classified according to the index to which the queries will be submitted.

Figure 4.5 shows clearly that the FR and BoW retrieval systems have much better than linear weak scalability. This is because as the size of the indexes increase, the number of comparisons that must be made increases at a much slower rate. For the FR system, as the size of the database increases, the time taken to classify the query features prior to being compared to the index takes slightly longer because the tree is larger. The comparison to the index takes on average the same amount of time no matter the size of the database. This explains why the throughput increases linearly with the number of machines added to the cluster. For the BoW system, since we fixed the number of leaf nodes to 1,000,000,

BoW and FR Throughput Results			
Nodes in Cluster	Database Images	Mean Throughput (μ)	Standard Deviation (σ)
FR Throughput Results			
5	500,000	0.89	0.094
10	1,000,000	1.62	0.015
20	2,000,000	2.89	0.067
BoW Throughput Results			
5	500,000	2.23	0.031
10	1,000,000	5.13	0.043
20	2,000,000	8.97	0.018

Table 4.10: BoW and FR Throughput Results

FR Throughput 95% Confidence Bounds			
Cluster Nodes	Database Images	Lower Limit	Upper Limit
10	1,000,000	1.58	1.66
20	2,000,000	2.72	3.06
BoW Throughput 95% Confidence Bounds			
Cluster Nodes	Database Images	Lower Limit	Upper Limit
10	1,000,000	5.02	5.23
20	2,000,000	8.93	9.01

Table 4.11: FR and Bow Throughput with 95% Confidence

the only change as the database grows is the number of multiplications that need to be done at each visual word represented in the query image. The throughput of the system is

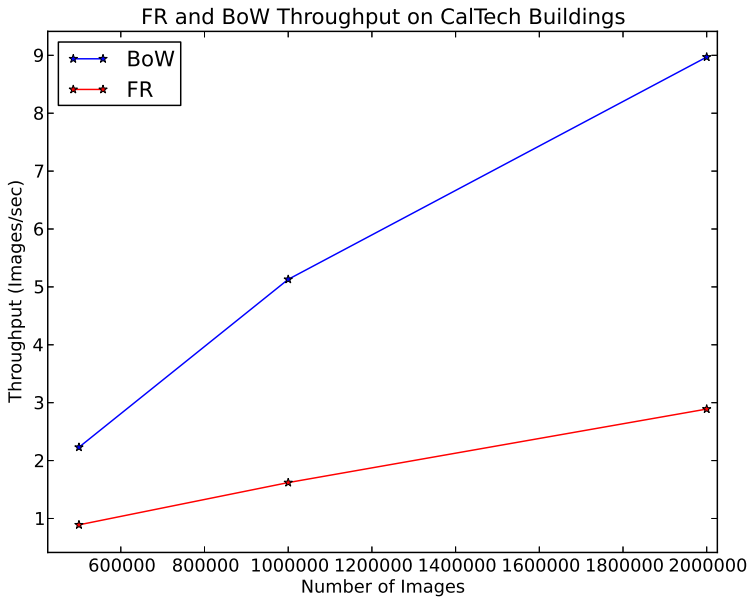


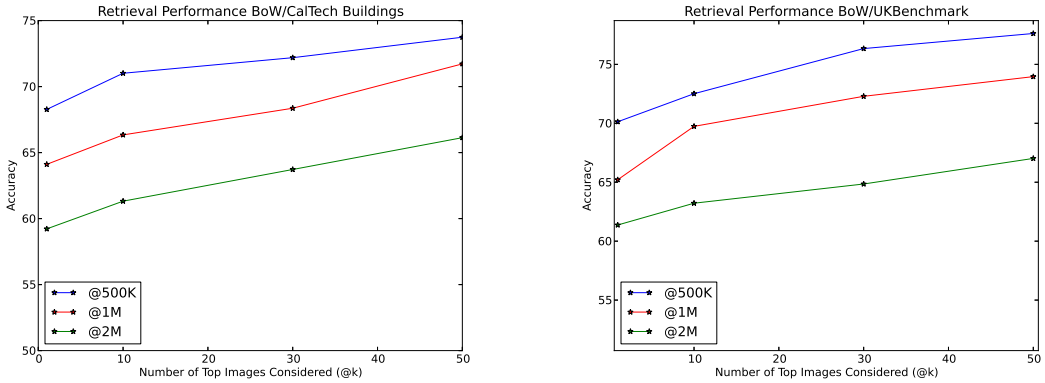
Figure 4.5: Weakly Scaled Throughput for FR and BoW Retrieval

also important, because it essentially represents client wait time. We have shown that the throughput for both types of image retrieval systems, has weak scalability much better than linear.

4.4 Retrieval Performance

Retrieval Performance is the last metric that we examine. We show retrieval performance at k, or the percentage of times that the correct ground truth image is returned in the top k results. We do not do any formal analysis on the retrieval performance, because our implementation has not changed any techniques that would directly affect retrieval performance. It is well documented that as the database image set increases, the retrieval performance suffers tremendously. This is intuitive: as there are more database images to get confused with the correct image, the retrieval algorithms get confused more often. We show performance at $n = [1, 10, 30, 50]$.

Figures 4.6a and 4.6b show the retrieval performance for the BoW retrieval system on the CalTech Buildings, and the UK Benchmark datasets respectively. We performed better across the board on the UK Benchmark dataset. The UK Dataset is very controlled, and there is very little in each image besides the object of focus. We believe this is why we consistently perform better on it.

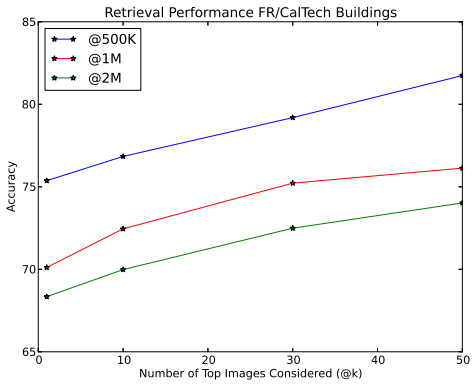


(a) CalTech Buildings

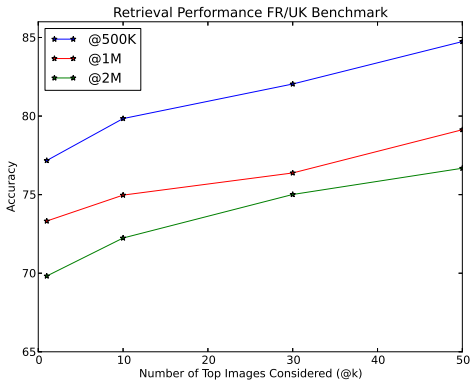
(b) UK Benchmark

Figure 4.6: Retrieval Performance for BoW System on Caltech Buildings and UK Benchmark Datasets at Multiple Precision Levels

Figures 4.7a and 4.7b show the retrieval performance for the FR retrieval system on the CalTech Buildings, and the UK Benchmark datasets respectively. We can see here that the FR system performs about 10% better than the BoW system across the board on both datasets. Recall, this improved performance comes at the cost of roughly an order of magnitude more storage for the FR HKM index, and nearly an order of magnitude increase in index creation time. While improving or evaluating the performance of the FR and BoW methods is not our focus, it is important to present the performance, since performance is the end deliverable of any image retrieval system.



(a) CalTech Buildings



(b) UK Benchmark

Figure 4.7: Retrieval Performance for FR System on Caltech Buildings and UK Benchmark Datasets at Multiple Precision Levels

V. Future Work and Conclusion

With the US military collecting more image data than ever, new techniques need to be created to glean useful information from this large amount of data. Image retrieval systems provide a way search through large collections of images for a particular image. BoW retrieval systems provide fast index creation, and low storage requirements with a sacrifice in retrieval accuracy. FR systems exhibit slower index creation, larger storage requirements, with better retrieval accuracy. Both types of systems provide the same service. When implemented with the big data processing power of MapReduce, we provide a way to extract the information contained in large amounts of image data.

While many image retrieval methods exist, as the size of database image collections increase, many of these methods falter. Traditional methods are plagued by issues of database storage, retrieval throughput, and index creation time. This research combines the scalable parallelization and data distribution of MapReduce with the highly researched, proven techniques of image retrieval in order to build a scalable image retrieval system. The objective of this thesis research is to build two linearly scalable image retrieval systems. In order to accomplish these goals, the following steps were taken:

1. Create and implement MapReduce Hierarchical K-Means BoW and FR indexing algorithms.

2. Develop MapReduce Query Image Comparison algorithms for BoW and FR Indexes.

3. Analyze the scalability of each step in the image retrieval system.

In order to definitively state whether the objective of developing two linearly scalable image retrieval systems was met, we broke the scaling of the systems into parts:

1. Verify linear scalability of FR HKM and Inverted Files index storage

2. Verify linear scalability of FR HKM and Inverted Files index creation

3. Verify linear scalability of FR and BoW retrieval throughput

4. Observe retrieval acuracy

Linear scalability of index storage was demonstrated for both BoW and FR HKM indexes. The scalability of FR and BoW index creation time were nearly linear. However, due to a randomness in cluster initialization and other overhead, index creation times for both systems are slightly worse than linear. Finally, the throughput of both the BoW and FR retrieval algorithms scaled far better than linear. This is very important since the throughput drives the amount of time a user would have to wait for a response. While we were not able to show linear scaling for either of the systems, we are satisfied by the scalability of the throughput.

5.1 Contributions

The novel contributions in this work include:

1. MapReduce implementation of HKM Inverted Files Index Creation

2. MapReduce implementation of HKM FR Index Creation

3. MapReduce implementation of tf-idf retrieval scheme for BoW image retrieval

4. MapReduce implementation of k-nearest neighbor retrieval scheme for FR image retrieval

5. Scalability analysis of HKM FR and HKM BoW image retrieval systems

This work has shown the first MapReduce implementations of both FR HKM and HKM BoW image retrieval systems. These systems have been demonstrated to be linearly scalable with respect to storage, index creation for BoW, and retrieval throughput. While

the BoW method was far superior in terms of index storage and creation times, the FR system yielded much better accuracy. This suggests that the type of image retrieval system should be chosen based on the needs of the application.

5.2 Extensions

There are many logical extensions to this research. While Hadoop is an ideal tool to solve problems dealing with large amounts of data, it is not optimal for iterative algorithms. Since HKM is an iterative algorithm, this is obvious area for improvement. Apache Mahout [37] is a software project, built on top of Hadoop, that has implemented scalable machine learning and clustering algorithms, such as K-Means. Utilizing these optimized algorithms could improve FR HKM index creation time considerably.

Retrieval accuracy, is also an area for potential improvements. This can only be improved by re-thinking the traditional image retrieval techniques. While implementing traditional image retrieval techniques on MapReduce allows for scalability in terms of index creation, retrieval throughput, and storage, just as in serial image retrieval systems, as the number of database images increases, the retrieval accuracy decreases. This leads to a system that is not scalable in terms of accuracy.

As was the case in this research, the scalability of MapReduce programs is typically limited only by physical resources. The Air Force is moving towards doing most resource on cloud infrastructures, to avoid costs of maintaining hardware and scaling problems such as this. Hadoop is very well suited to cloud computing, and extending this research to a cloud environment to run larger experiments, is also a logical extension. This research could be extended into practical military applications. If set up in a cloud environment, an image retrieval system built like this could be used to recognize many different types of objects from anywhere. A deployed soldier could easily snap a picture of an object with a cell phone, and submit that image to the cloud environment. Within seconds, that object is identified and information associated with it can be served.

5.3 Conclusion

In conclusion, this research provides two methods for performing large scale image retrieval. While we were unable to claim linear scalability of the overall system, the on-line phase scales much better than linearly, and the off-line phase is a one time cost. The methods presented here do provide a method for processing large amounts of data, however handling large amounts of data still proves challenging. Now that multiple scalable image retrieval systems have been demonstrated, image retrieval techniques must be adapted to provide better accuracy. Finally, in order to compare new image retrieval implementations and techniques, a cloud-based research environment should be proposed for standardization of future experiments.

Bibliography

[1] David Abrahams. Building hybrid systems with boost.python, March 2003.

[2] S. Agarwal, N. Snavely, I. Simon, S.M. Seitz, and R. Szeliski. Building rome in a day. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 72–79, 2009.

[3] Mohamed Aly, Mario Munich, and Pietro Perona. Bag of words for large scale object recognition. *computational vision lab, Caltech, Pasadena, CA, USA*, 2011.

[4] Mohamed Aly, Mario Munich, and Pietro Perona. Distributed kd-trees for retrieval from very large image collections. In *British Machine Vision Conference, Dundee, Scotland*, 2011.

[5] Mohamed Aly, Mario Munich, and Pietro Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *Applications of Computer Vision (WACV), 2011 IEEE Workshop on*, pages 418–425. IEEE, 2011.

[6] Mohamed Aly, Peter Welinder, Mario Munich, and Pietro Perona. Towards automated large scale discovery of image families. In *Computer Vision and Pattern Recognition Workshops, 2009. CVPR Workshops 2009. IEEE Computer Society Conference on*, pages 9–16. IEEE, 2009.

[7] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[8] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer Vision–ECCV 2006*, pages 404–417. Springer, 2006.

[9] Dhruba Borthakur. The hadoop distributed file system: Architecture and design, 2007.

[10] G. Bradski. *Dr. Dobb's Journal of Software Tools*, 2000.

[11] Terry Costlow. Big data poses big challenge for military intelligence, 2012.

[12] Gabriella Csurka, Christopher Dance, Lixin Fan, Jutta Willamowski, and Cédric Bray. Visual categorization with bags of keypoints. In *Workshop on statistical learning in computer vision, ECCV*, volume 1, page 22, 2004.

[13] Anders Lindbjerg Dahl, Henrik Aanæs, and Kim Steenstrup Pedersen. Finding the best feature detector-descriptor combination. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2011 International Conference on*, pages 318–325. IEEE, 2011.

[14] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[15] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys (CSUR)*, 40(2):5, 2008.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[18] Bo Dong, Jie Qiu, Qinghua Zheng, Xiao Zhong, Jingwei Li, and Ying Li. A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by powerpoint files. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 65–72. IEEE, 2010.

[19] Friedrich Fraundorfer and Horst Bischof. Evaluation of local detectors on non-planar scenes. In *Proceedings of the Austrian Association for Pattern Recognition Workshop*, pages 125–132, 2004.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[21] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK, 1988.

[22] Amazon Web Services Inc. Amazon elastic map reduce, 2013.

[23] Andrew Edie Johnson. *Spin-images: a representation for 3-D surface matching*. PhD thesis, Citeseer, 1997.

[24] Bela Julesz. Textons, the elements of texture perception, and their interactions. *Nature*, 1981.

[25] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 78–85. ACM, 2010.

[26] S.P. Lloyd. Least squares quantization in pcm's. In *Bell Telephone Laboratories Paper*. Murray Hill, 1957.

[27] DavidG. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[28] Jiri Matas, Ondrej Chum, Martin Urban, and Tomás Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing*, 22(10):761–767, 2004.

[29] Krystian Mikolajczyk and Cordelia Schmid. Scale & affine invariant interest point detectors. *International journal of computer vision*, 60(1):63–86, 2004.

[30] Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(10):1615–1630, 2005.

[31] Krystian Mikolajczyk, Tinne Tuytelaars, Cordelia Schmid, Andrew Zisserman, Jiri Matas, Frederik Schaffalitzky, Timor Kadir, and Luc Van Gool. A comparison of affine region detectors. *International journal of computer vision*, 65(1-2):43–72, 2005.

[32] Diana Moise, Denis Shestakov, Gylfi Gudmundsson, and Laurent Amsaleg. Indexing and searching 100m images with map-reduce. In *Proceedings of the 3rd ACM conference on International conference on multimedia retrieval*, pages 17–24. ACM, 2013.

[33] Greg Mori, Xiaofeng Ren, Alexei A Efros, and Jitendra Malik. Recovering human body configurations: Combining segmentation and recognition. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–326. IEEE, 2004.

[34] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP (1)*, pages 331–340, 2009.

[35] Henning Müller, David M Squire, Wolfgang Mueller, and Thierry Pun. Efficient access methods for content-based image retrieval with inverted files. In *Photonics East'99*, pages 461–472. International Society for Optics and Photonics, 1999.

[36] David Nister and Henrik Stewenius. Scalable recognition with a vocabulary tree. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2161–2168. IEEE, 2006.

[37] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in action*. Manning, 2011.

[38] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.

[39] Jeffrey Shafer, Scott Rixner, and Alan L Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems &*

Software (ISPASS), 2010 IEEE International Symposium on, pages 122–133. IEEE, 2010.

[40] Denis Shestakov, Diana Moise, Gylfi Thór Gudmundsson, Laurent Amsaleg, et al. Scalable high-dimensional indexing with hadoop. In *CBMI—International Workshop on Content-Based Multimedia Indexing*, 2013.

[41] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[42] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 1470–1477. IEEE, 2003.

[43] Stephen M Smith and J Michael Brady. Susana new approach to low level image processing. *International journal of computer vision*, 23(1):45–78, 1997.

[44] Tinne Tuytelaars and Krystian Mikolajczyk. Local invariant feature detectors: a survey. *Foundations and Trends® in Computer Graphics and Vision*, 3(3):177–280, 2008.

[45] Tinne Tuytelaars and Luc J Van Gool. Wide baseline stereo matching based on local, affinely invariant regions. In *BMVC*, volume 412, 2000.

[46] Brandyn White, Tom Yeh, Jimmy Lin, and Larry Davis. Web-scale computer vision using mapreduce for multimedia data mining. In *Proceedings of the Tenth International Workshop on Multimedia Data Mining*, page 9. ACM, 2010.

[47] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[48] Jun Yang, Yu-Gang Jiang, Alexander G Hauptmann, and Chong-Wah Ngo. Evaluating bag-of-visual-words representations in scene classification. In *Proceedings of the international workshop on Workshop on multimedia information retrieval*, pages 197–206. ACM, 2007.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2014	Master's Thesis	Sept 2012–Mar 2014

4. TITLE AND SUBTITLE

Large Scale Hierarchical K-Means Based Image Retrieval With MapReduce

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Murphy, William E., Second Lieutenant, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-14-M-56

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Image retrieval remains one of the most heavily researched areas in Computer Vision. Image retrieval methods have been used in autonomous vehicle localization research, object recognition applications, and commercially in projects such as Google Glass. Current methods for image retrieval become problematic when implemented on image datasets that can easily reach billions of images.

In order to process these growing datasets, we distribute the necessary computation for image retrieval among a cluster of machines using Apache Hadoop. While there are many techniques for image retrieval, we focus on systems that use Hierarchical K-Means Trees. Successful image retrieval systems based on Hierarchical K-Means Trees have been built using the tree as a Visual Vocabulary to build an Inverted File Index and implementing a Bag of Words retrieval approach, or by building the tree as a Full Representation of every image in the database and implementing a K-Nearest Neighbor voting scheme for retrieval. Both approaches involve different levels of approximation, and each has strengths and weaknesses that must be weighed in accordance with the needs of the application. Both approaches are implemented with MapReduce, for the first time, and compared in terms of image retrieval precision, index creation run-time, and image retrieval throughput. Experiments that include up to 2 million images running on 20 virtual machines are shown.

15. SUBJECT TERMS

Image Retrieval, MapReduce, Hierarchical K-Means, Big Data, Hadoop

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Kennard R. Laviers, (AFIT/ENG)
U	U	U	UU	87	19b. TELEPHONE NUMBER *(include area code)* (937) 255-6565 x4395 Kennard.Laviers@afit.edu